University of Zurich[UZH]

Department of Informatics

Efficient Processing and Reasoning of Semantic Streams

Dissertation submitted to the Faculty of Business,
Economics and Informatics
of the University of Zurich

to obtain the degree of
Doktor / Doktorin der Wissenschaften, Dr. sc.
(corresponds to Doctor of Science, PhD)

presented by
Shen Gao
from China

approved in Feb 2018

at the request of
Prof. Abraham Bernstein, Ph.D.
Prof. Jeff Z. Pan, Ph.D.

University of Zurich[UZH]

The Faculty of Business, Economics and Informatics of the University of Zurich hereby authorizes the printing of this dissertation, without indicating an opinion of the views expressed in the work.

Zurich, Feb 14, 2018

Chairman of the Doctoral Board: Prof. Dr. Sven Seuken

Abstract

The digitalization of our society creates a large number of data streams, such as stock tickers, tweets, and sensor data. Making use of these streams has tremendous values. In the Semantic Web context, live information is queried from the streams in real-time. Knowledge is discovered by integrating streams with data from heterogeneous sources. Moreover, insights hidden in the streams are inferred and extracted by logical reasoning.

Handling large and complex streams in real-time challenges the capabilities of current systems. Therefore, this thesis studies how to improve the efficiency of processing and reasoning over semantic streams. It is composed of three projects that deal with different research problems motivated by real-world use cases. We propose new methods to address these problems and implement systems to test our hypotheses based on real datasets.

The first project focuses on the problem that sudden increases in the input stream rate overload the system, causing a reduced or unacceptable performance. We propose an eviction technique that, when a spike in the input data rate happens, discards data from the system to ensure the response latency at the cost of a lower recall. The novelty of our solution lies in a data-aware approach that carefully prioritizes the data and evicts the less important ones to achieve a high result recall.

The second project studies complex queries that need to integrate streams with remote and external background data (BGD). Accessing remote BGD is a very expensive process in terms of both latency and financial cost. We propose several methods to minimize the cost by exploiting the query and the data patterns. Our system only needs to retrieve data that are more critical to answer the query and avoids wasting resources on the remaining data in BGD.

Lastly, as noise is inevitable in real-world semantic streams, the third project investigates how to use logical reasoning to identify and exclude the noise from high-volume streams. We adopt a distributed stream processing engine (DSPE) to achieve scalability. On top of a DSPE, we optimize the reasoning procedures by balancing the costs of computation and communication. Therefore, reasoning tasks are compiled into efficient DSPE workflows that can be deployed across large-scale computing clusters.

Zusammenfassung

Die Digitalisierung unserer Gesellschaft produziert massenweise Streams, wie beispielsweise Aktienkurse, Tweets und Sensordaten. Die Nutzung dieser Streams bringt enorme Vorteile mit sich. Im Kontext des Semantic Web wird Live-Information aus den Streams in Echtzeit abgefragt. Durch Integration mehrerer Streams aus unterschiedlichen Quellen kann Wissen entdeckt werden. Durch logische Schlüsse können verborgene Erkenntnisse aus den Streams gewonnen werden.

Bestehende Systeme sind mit der zeitnahen Verarbeitung von grossen und komplexen Streams herausgefordert. Diese Arbeit untersucht Effizienzsteigerungen von Prozessierung und Schlussfolgerung in semantischen Streams. Sie besteht aus drei Projekten, welche unterschiedliche, aus der Praxis entnommene Forschungsfragen adressieren. Wir schlagen neue Methoden vor um diese anzugehen und implementieren Systeme zum testen unsere Hypothesen aufgrund realer Datensätze.

Das erste Projekt konzentriert sich auf sprunghafte Anstiege der Eingangsrate, welche das System überlasten, und somit dessen Leistung bis in einen untragbaren Bereich herabsetzen können. Wir schlagen eine Methode vor, welche gezielt Daten aus dem System entfernt, um die Latenz auf Kosten des Recalls aufrechtzuerhalten. Neu an dieser Lösung ist der datenbezogene Ansatz, welcher Daten sorgfältig priorisiert und weniger wichtige Daten zuerst aussortiert, um den Recall des Resultates möglichst hoch zu halten.

Im zweiten Projekt studierten wir komplexe Anfragen, welche nur durch eine Kombination von Stream und zusätzlichen, entfernt gespeicherten Hintergrundinformation (HGI) beantwortet werden können. Zugriff auf HGI ist ein ebenso teurer wie zeitaufwändiger Prozess. Wir schlagen mehrere Methoden vor, welche Muster in Daten und Anfrage nutzen um die Kosten zu minimieren. Dadurch braucht unser System bloss Daten abzugreifen, welche für die Beantwortung der Anfrage kritisch sind, ohne wertvolle Ressourcen mit der Verarbeitung der restlichen HGI zu verschwenden.

Letztendlich, da Rauschen in Streams unvermeidbar ist, untersucht das dritte Projekt, wie durch logisches Schlussfolgern Rauschen identifiziert und es aus des Verarbeitung ausgeschlossen werden kann. Eine verteilte Stream Processing Engine (DSPE) wurde als Grundlage angenommen, um Skalierbarkeit zu gewährleisten. Darauf aufbauend optimierten wir die Reasoning-Prozedur, in dem Kommunikation und Berechnung gegeneinander ausbalanziert werden. Dadurch werden Reasoning-Aufgaben als effiziente Abläufe formuliert, welche auf gross angelegten Rechenclustern eingesetzt werden können.

Acknowledgements

First of all, I want to express my sincere gratitude to my supervisor Prof. Abraham Bernstein (Avi) for the opportunity of allowing me to pursue a Ph.D. at the University of Zürich. He granted me great research freedom to explore the topics that I am interested in. His continuous support and guidance have taught me so many things. His contribution in terms of time, ideas, and mental support has made my study productive and fun.

I would like to extend my gratitude to my co-examiner Dr. Jeff Z. Pan for his time and invaluable feedback on my research. I am also very grateful for his contributions as a co-author on one of the papers in the thesis.

I am fortunate to have learned a lot from my collaborators during my study. I would like to thank Dr. Daniele Dell'Aglio who devoted enormous effort to help me. The same thanks also go to my other collaborators and co-authors: Soheila Dehghanzadeh, Emanuele Della Valle, Alessandra Mileo, Thomas Scharrenbach, Jörg-Uwe Kietz, Michael Feldman, Marc Novel, and Katerina Papaioannou.

During the time at the Department of Computer Science (IFI), I met so many wonderful friends and colleagues. They have made my study very enjoyable. Thanks to all my friends: Bibek, Cosmin, Daniel (Spicar), Daniel (Strebel), Ela, Helen, Lorenz, Marc, Matthias, Patrick, Pengcheng, Philip, Timo, Tobias, and Wen.

Finally, thanks to my family's love and understanding. Without my wife Yang's support, this dissertation would not have been possible.

Table of Contents

Part I

Synopsis

1 Introduction

The digitalization of every aspect of our society brings a huge number of diverse data streams. People are continuously producing an enormous amount of streams on the Web. Today, thousands of tweets are written every second. Hours of video are uploaded to YouTube every minute. The rise of the Internet of Things (IoT) creates large amounts of high-speed streams from millions of intelligent devices [7]. Under the Smart City concept, widely-deployed sensor networks are producing real-time information about traffic, weather, power grid, and other topics [67]. Scientific research also creates streams at an unprecedented speed. The world's largest particle physics lab, CERN, is generating petabyte-scale data every second[1].

Processing these streams in a timely manner has tremendous benefits. Our daily life has been improved by the real-time information. For example, live traffic information reduces people's commuting time by providing up-to-the-minute travel plans. In some cases, processing streams promptly can save people's lives. An earthquake monitoring system has large amounts of sensors deployed to monitor environmental and geological conditions. Their streams are processed as fast as possible to let people get immediate notification when earthquakes happen [81]. In this example, real-time processing is critical. Sending the notification earlier gives people a better chance to survive.

Integrating streams from different sources also has great value. For example, sensor data is coupled with social media streams to forecast smog-related health hazards [21]. The value of processing data from heterogeneous sources becomes even more prominent in the context of Semantic Web. The Linked Open Data (LOD) makes it possible to enrich streams with various kinds of background knowledge [15, 78].

Reasoning over streams can discover results with more insights. In Semantic Web, deductive reasoning techniques make hidden information explicit. Authors in [75] argue that reasoning over large amounts of semantic streams can support the decision process of extremely large numbers of concurrent users in real time.

Given its importance, this thesis considers the problem of handling streams in the Semantic Web context. The challenges of our study can be summarized by three Vs of "big data"[2]: Velocity, the rapidly changing nature of the streams requires real-time processing; Variety, heterogeneous semantic streams and background data need to be integrated; Volume, the sheer amount of the streams requires a distributed solution, since no single computer is powerful enough to process them. As current systems cannot fully address these challenges, we, therefore, pose the central question that this thesis attempted to answer:

How to improve the efficiency of processing and reasoning over semantic streams?

[1] https://cacm.acm.org/news/110048-cern-experiments-generating-one-petabyte-of-data-every-second/fulltext
[2] http://www.ibmbigdatahub.com/infographic/four-vs-big-data

2 Background

To refine this overall question, it is of advantage to have some background introduction, which we present in this section. It introduces semantic streams and related processing and reasoning techniques. It also presents an overview of distributed stream processing engines.

The following is a running example of the semantic stream considered in this section.

> **Example 1 (Semantic streams)** *Consider an IPTV application that has a stream describing user behavior [36]:*
>
> $$(\mathsf{User1},\ \mathsf{joins},\ \mathsf{ChannelA}, 1)$$
> $$(\mathsf{User2},\ \mathsf{joins},\ \mathsf{ChannelB}, 3)$$
> $$(\mathsf{User1},\ \mathsf{leaves},\ \mathsf{ChannelA}, 5)$$
>
> *Each data element in the stream represents a user joining or leaving a TV channel at a time point (e.g.,* User1 *joins* ChannelA *at time point* 1 *and leaves at time point* 5*). By exploiting the stream, the application can answer questions like: "which channels have over 1000 viewers in the past five minutes?". It can also infer implicit information by combining this stream with other streams and/or background data. For example, assuming we know that:*
>
> $$(\mathsf{ChannelA},\ \mathsf{hasLanguage},\ \mathsf{German})$$
>
> *If* User1 *often watches* ChannelA*, the system is possible to infer that:*
>
> $$(\mathsf{User1},\ \mathsf{speaks},\ \mathsf{German})$$

2.1 Semantic Streams

As shown in Example 1, the semantic stream considered in this thesis is rooted in the Resource Description Framework (RDF) data model of the Semantic Web. The Semantic Web is an extension of the World Wide Web (WWW) with RDF as the standard data interchange model [23]. In the RDF model, anything in the real world can be called a *resource* and denoted by Internationalized Resource Identifiers (IRIs) or literals. An RDF statement describes a relationship (called predicate) that holds between two resources (called subject and object, respectively). An RDF statement has the form of a triple (subject, predicate, object). In Example 1, (ChannelA, hasLanguage, German) is an RDF statement that represents the relationship hasLanguage between resources ChannelA and German. The standard query language of RDF data is named SPARQL [41]. A SPARQL endpoint is a service where users can query RDF data using SPARQL.

An RDF stream extends the model by considering the temporal dimension. It has the form of $S = ((d_1, t_1), \ldots, (d_n, t_n), \ldots)$, which is a potentially unbounded sequence of timestamped informative units (d_i, t_i) ordered by t_i. For each data item d_i, t_i is a timestamp (e.g., the three statements in Example 1 are at time points 1, 3, and 5). As

in [14, 19, 51], we consider timestamps as discrete. Specially, d_i is an RDF statement (e.g., the first three fields in each data element of Example 1 form an RDF statement). This thesis considers semantic streams that are serialized as RDF streams, as shown by the user behavior stream in Example 1.

2.2 Processing and Reasoning over Semantic Streams

RDF Stream Processing (RSP) performs various data manipulations over semantic streams, such as querying, aggregating, and transforming [28]. Many recent efforts have been devoted to this research topic. They adapt existing Semantic Web technologies to process RDF streams. Regarding the query language, SPARQL has been extended to several languages/systems, such as C-SPARQL [14], CQELS [51], and MorphStream [19]. These solutions share a common feature: they adopt the WINDOW operation to create time-varying views over the input RDF streams. Specifically, a WINDOW operation caches a portion of the most recent stream as a view, which can be processed by standard operators defined in SPARQL. Referring to Example 1, a WINDOW with a length of `4` time units at time point `6` contains both statements at timestamps `3` and `5`.

Reasoning, in the Semantic Web context, is the process of inferring logical consequences from a set of asserted facts. This process can discover information that is not explicitly stated in an ontology or a knowledge base. It can be performed for a variety of purposes like detecting inconsistencies or classifying data [80]. As with the development of RSP technologies, various reasoning techniques have been proposed to cope with semantic streams. For example, authors in [68] consider the case of reasoning over a dynamic ontology and define it as an *evolving ontology*. Some studies in this thesis also borrow this definition.

2.3 Distributed Stream Processing Engines (DSPEs)

Most existing RSP technologies are developed for a centralized system, which is not scalable to multiple computing nodes. For this reason, some of our studies employ DSPEs to process large amounts of streams in a distributed fashion. DSPEs were designed to process generic streams in clusters and cloud services. One of the first popular DSPEs is Storm [79], initially proposed by Twitter and today maintained as an Apache project. Storm relies on the notion of *topology*, which is a user-defined processing workflow. It instantiates a conceptual topology to multiple computing tasks and distributes them to different nodes. Twitter recently proposed Heron [48] as a successor of Storm. Heron addresses some shortcomings of Storm, such as limited performance monitoring, difficulties of deploying and debugging in clusters with heterogeneous nodes. Other DSPEs are Apache Flink [20], Apache Samza [86], and Google MillWheel [2]. Each of these systems has a different design goal. For example, Apache Samza uses a key-value store to save and query processing states among nodes. Apache Flink has the advantage of combining batched-based and stream-based processing. Google MillWheel can cope with data that arrive out-of-order in the stream. Almost all of them share the same idea of letting the user define a workflow of operators, similar to a Storm's topology.

3 Problem Statement

As mentioned in Section 1, the challenges of handling semantic streams can be summarized by the three **V** of "Big Data". In the following, we expand the three **V** into six challenges that lead to our central question. Each research project in this thesis aims at tackling one or several of these challenges.

Challenge 1. Ensure reactiveness. Unlike processing static data, a stream processing system must be reactive. Time-critical applications have to continuously process the incoming streams and provide answers within a latency threshold. Usually, the latency is strictly constrained to real-time (sub-seconds) or near real-time (minutes) [29]. The latency threshold has to be fulfilled under any circumstances. Sometimes, providing answers in a timely fashion is even preferred over providing the entire and accurate result [77]. Therefore, a semantic stream system always has to guarantee reactiveness.

Challenge 2. Handle workload spikes. In real-world applications, the incoming stream rate fluctuates over time. When a spike occurs in the input stream rate, the processing workload can be orders of magnitude higher than normal. The stream rate, however, is often difficult to predict in advance, which makes it impossible to decide how many processing resources the system needs [10, 24]. This challenge requires RSP systems to handle peak workloads with limited resources during runtime.

Challenge 3. Combine streams with background data (BGD). To support complex queries, semantic streams are usually combined with BGD, such as a knowledge base hosted by a SPARQL endpoint on the Web. Querying BGD to enrich the streams allows users to harvest more insightful results [78]. For example, financial analysis queries usually need to combine stock tickers with company profiles. Therefore, an RSP system needs to handle both streams and BGD at the same time, to support complex queries.

Challenge 4. Acquire BGD efficiently. Closely related to Challenge 3, the cost of acquiring BGD can be prohibitively high in terms of both accessing time and money. Retrieving a large amount of BGD online takes a long time. It can be infeasible to access every required BGD element, when given a response-time constraint [34]. Furthermore, some BGD data providers may charge money for data access [26]. Therefore, we need to carefully plan the BGD accesses to avoid excessive acquiring costs.

Challenge 5. Handle high volume. Streams can arrive in a very high volume that is too large to fit into one computing node. As mentioned in Section 2.3, many DSPEs are proposed to process generic streams in a distributed fashion. However, how to employ such a DSPE to process large amounts of semantic streams still needs in-depth investigations.

Challenge 6. Exhibit robustness with noise. Noise is inevitable in real-world streams. Some simple noise is relatively easy to be found. For example, using data range filters can exclude outliers. However, semantic streams usually come with complex schemas. Implicit information hidden in the stream may violate the semantics of the underlying conceptual model. Although logical reasoning can help to solve this problem, efficiently reasoning over semantic streams remains a difficult problem.

4 Research Questions and Hypotheses

This section presents the three research questions (RQs) addressed in the three papers presented in Part II. For each RQ, we first introduce its motivation and give a concrete use case. Then, we derive and discuss the main hypotheses.

As an overview, Figure 1 illustrates an RSP system that takes semantic streams as input and produces output continuously. The three RQs share the goal of improving the overall system efficiency, while each RQ aims at one specific aspect of the system. On the left in Figure 1, RQ1 deals with spikes in the input stream. On the top, RQ2 optimizes the cost of access external BGD. RQ3 (on the right) develops a method that performs reasoning over semantic streams in a distributed fashion.

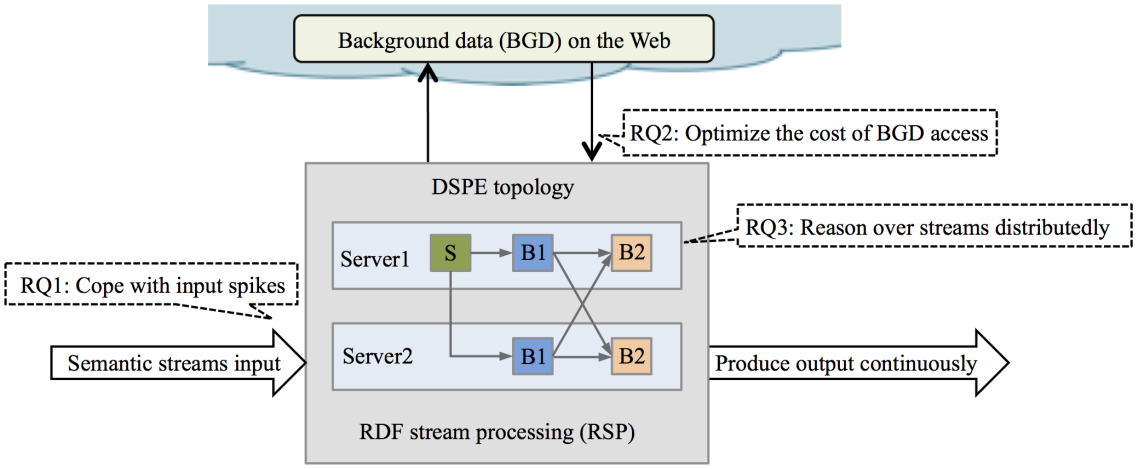


Fig. 1: The three RQs share the goal of improving the overall system efficiency, while each RQ aims at one specific aspect of the system.

4.1 RQ 1: How can we use an eviction strategy to cope with the problem that a workload spike overwhelms the system?

As discussed in Challenges 1 and 2 of Section 3, stream systems must guarantee a constant response latency at all times. However, a sudden increase in the stream rate can abruptly lead to a heavy workload that overwhelms the system processing capacity.

> **Example 2 (Workload spikes)** *Consider an IPTV application, the number of viewers in a news channel can significantly fluctuate over time [35]. When breaking news are being reported, the number of viewers will grow unexpectedly. Real-time applications that analyze viewer behaviors must be able to cope with this spike in the workload.*

Current methods handle this situation by limiting the scope of a query with a time window, as explained in Section 2.2. The window operation, however, does not solve the problem completely. It is still possible that a high stream data rate makes the window content too large to be processed within a given latency threshold. An eviction strategy can help solve the problem by selecting data based on some criteria and deleting them from those windows. It does so at the cost of introducing potential errors: mistakenly evicting data will lead to a lower result recall. Therefore, we asked the question of how to design an eviction strategy that can handle the workload spikes without significantly deteriorating the result recall.

When developing such an eviction strategy, we first need to understand existing methods such as Random, First-In-First-Out (FIFO), and Least Recently Used (LRU). These methods are data-agnostic. They may cause poor result recalls since data are indiscriminately chosen for eviction. To address this problem, we design a novel data-aware eviction strategy. The strategy uses a dynamic ranking mechanism to identify important data items according to the stream. These data must also be stored for a longer time than others. Above proposition covers the following hypothesis:

Hypothesis 1: A data-aware eviction strategy can improve the result recall when a workload spike overwhelms the system processing capability.

4.2 RQ 2: How can we plan the access of remote background data to improve the result freshness for complex queries?

Challenge 3 in Section 3 shows that complex queries need to handle the join between streaming data and BGD. However, acquiring BGD can be very costly, since the target BGD is oftentimes large, maintained externally, and changing slowly, as discussed in Challenge 4 of Section 3.

> **Example 3 (Expensive BGD accesses)** *Consider the use case that a company wants to persuade influential users on social networks to post commercial endorsements. The company sets the criteria for the influential users as follows. First, they must be trendsetters with mentions of more than 1,000 posts in the past 20 minutes. Second, they must be important, i.e., they have more than 10,000 followers on their profile [26]. A complex query can identify those users by joining the post stream with the user profiles. However, querying user profiles is an expensive procedure. Each access may cost a certain amount of money. When the number of followers changes over time, it even requires frequently accessing the user profiles to get the most recent update.*

To reduce this cost, practical applications usually maintain a local view to cache the relevant BGD. As BGD can change slowly in the remote service, the local view has to be updated repeatedly to avoid stale data leading to wrong answers. However, accessing BGD is usually subject to some realistic update-budget constraints. For example, the number

of accesses within a time window cannot exceed a limit. Therefore, we investigated how to allocate an update budget to improve the result freshness.

To study this question, we first need to find a mathematical model to formally define the problem and formulate the optimization goal. Second, it is critical to understand how data and query patterns affect the result freshness over time so that the budget can be dynamically distributed. Third, different sub-query patterns require different optimization goals, the maintenance process must be able to handle different sub-query cases. In the context of these requirements, we investigated the hypothesis:

Hypothesis 2: A maintenance process that allocates the update budget according to data and query patterns can improve the result freshness.

4.3 RQ 3: How can we use logical reasoning to check the consistency of large amounts of semantic streams w.r.t a conceptual model?

As discussed in the Challenges 5 and 6 of Section 3, semantic noise in streams is very complex and hard to find. This is because the streams are usually defined by an underlying conceptual model. The noise can implicitly violate constraints set by the model. Existing reasoning techniques can spot this kind of noise on static data. However, reasoning over streams is a difficult task because the system needs an incremental method to handle newly arrived data. The question becomes even more challenging when reasoning over large amounts of streams.

> **Example 4 (Complex noise)** *Consider a large e-commerce website that maintains its product catalog as a Knowledge Base (KB). Each product is represented as an entity with attributes describing its features. Thousands of sellers generate a huge amount of new entities every minute. However, a deceitful seller may intentionally provide fake product information. The company is interested in checking whether a newly added entity is consistent with an apriori model about the product. For example, a conceptual model of smart phones asserts that "iPhone" and "AndroidPhone" are disjoint classes. When a seller adds a new entity "xPhone" with exclusive attributes that can be inferred as an instance of both classes, the system should spot the inconsistency and stop adding it to the KB.*

Existing reasoning tasks are mostly based on static data. We first need to adapt them to a dynamic setting. Given the huge volume of streams, employing a DSPE seems to be a good solution to improve scalability. However, it is still a challenging problem to perform stream reasoning on top of a DSPE. As discussed in Section 2.3, these engines require users to define a processing topology. Therefore, we need to study how to build and optimize reasoning topologies that can ensure the streams are confirmed with a given complex schema. This leads to the third hypothesis of this thesis:

Hypothesis 3: A DSPE topology, which implements reasoning procedures, can efficiently check the consistency of large amounts of semantic streams w.r.t a conceptual model.

5 Contributions

This section gives an overview of the main outcomes of each project. The details of our solutions and results can be found in the corresponding paper in Part II.

5.1 The Clock Data-Aware Eviction Approach: Towards Processing Linked Data Streams with Limited Resources

To answer RQ 1, we first studied two datasets to demonstrate the problem that stream-rate bursts can overload a system. We used the well-established SR-Bench dataset [85], as well as the real-world ViSTA-TV[3] dataset that comes from the IPTV domain. Our study discovered that in both cases, RSP systems suffer from stream-rate fluctuations. The peak workload can easily overflow the working memory allocated to the system. For example, in the ViSTA-TV dataset, the start/end times of popular TV shows can lead to a sudden change in the rate of a user behavior stream. When many users join a channel at the same time, it quickly exhausts the working memory.

To verify Hypothesis 1, we implemented and tested several data-agnostic eviction approaches, such as Random, FIFO, and LRU. Results showed that these strategies save memory usage but cause a low result recall since they naïvely choose data to evict. Some data items that are expected to produce more results were evicted too early. Therefore, we proposed our eviction strategy named Clock. In Clock, each data item in the memory is associated with a score, which prioritizes the data during an eviction. The score increases when the data produces more results and decreases with the time that it stays in memory. In this way, the score is dynamically adjusted according to the stream. It estimates the likelihood of producing future results based on the performance history and retains productive entries for a longer time. A variant of Clock is also proposed, where a tunable parameter can control the weight of increasing and decreasing the score.

Experimental results showed that existing data-agnostic approaches can curb the memory consumption of workload spikes with a low result recall. Clock and its variant outperform the often-used LRU and FIFO strategies by factors between 1.5 and almost 3.

5.2 Planning Ahead: Stream-Driven Linked-Data Access under Update-Budget Constraints

To answer RQ 2, we first demonstrated that the budget in terms of allowed number of remote BGD accesses is usually very limited. In real scenarios, SPARQL endpoints are exposed over the Internet, and each request can take more than $500ms$ [60]. Given a very strict response time, the amount of possible accesses can be insufficient (e.g., a five-second response time only accommodates for at most ten sequential accesses).

As mentioned in Section 4.2, practical systems usually adopt a local view to cache relevant BGD. We defined that a *maintenance process* as the process of refreshing a portion of the local view. It decides which is the most desirable part of the local view to refresh under a given update budget. The budget is expressed in terms of the number of

[3] http://vista-tv.eu/

accesses. We found that a bipartite graph can accurately model this problem. Under this model, data in the stream and the stale data in BGD form the vertices of the two disjoint sets; the join relationships between them create the edges. Then, the selection problem is converted into the problem of choosing a minimal set of BGD vertices to update so that it achieves the best response freshness, where the freshness is defined as the ratio of the number of up-to-date results over the total number of results in a window evaluation. We noticed that this optimization goal of achieving best response freshness could change according to the type of sub-query patterns. A basic solution named Selectivity-Based Maintenance (SBM) was proposed for two different sub-query patterns. When a query involves a Basic Graph Pattern (BGP) over the background data, a result corresponds to an edge in the graph. In this sub-query pattern, SBM is proven to be a locally optimal solution, which leverages the selectivity of the BGD data. It selects the BGD data with the highest number of edges so that it produces maximal amount of fresh results. When the sub-query is an aggregation over the BGD, the optimization goal is changed to achieve maximal amount of aggregated results. A fresh aggregated result can only be produced when a data item in the stream has all its associated edges updated. We modeled this problem with integer programming and showed that it is an NP-hard problem. SBM for the aggregation sub-query pattern is a heuristic algorithm that finds the stream data with the highest selectivity and updates their corresponding BGD.

Also, two extensions were proposed to improve SBM: the first one exploited the sliding window operation, which can be used to estimate how long a streamed data item will stay in the system. Based on this idea, our proposed algorithm, named Impact-Based Maintenance (IBM), selects data items with longer impact. The impact is defined as the total number of expected results, which can be precisely calculated over the sliding window. Second, when the budget is allocated for a relatively longer time (multiple windows), we proposed a solution named Flexible Budget Allocation (FBA). FBA not only optimizes the current budget but also considers future ones. Therefore, it dynamically allocates an appropriate amount of budget to each query evaluation.

Lastly, our solutions are implemented in a real RSP engine, C-SPARQL. Experiments showed that our methods could efficiently allocate a given update budget and improve the result freshness by up to 93% over baseline algorithms such as Random Selection or Least Recently Updated.

5.3 Distributed Stream Consistency Checking

To address RQ 3, we explored how to use logical reasoning to check the consistency of the stream w.r.t. a conceptual model. We made the following three contributions.

Our first contribution is to formally define the consistency-checking task as a stream reasoning problem. Existing reasoning tasks are mainly based on a static ontology. We modeled the problem by adopting the evolving ontology notion based on the logic DL-lite$_{core}$ [16, 68, 82], where the TBox is static, and the ABox assertions arrive as a stream. A window operation captures a portion of the ABox stream. The consistency checking task is performed over the window content against the TBox continuously.

We designed algorithms to compile a TBox into consistency checking procedures that can be executed as a DSPE topology. We first proposed a baseline method named Negative-Inclusion Topology Method (NTM). NTM exhaustively compiles the TBox closure into checking queries and registers them in a DSPE topology. This approach has the shortcoming of generating an excessive amount of checking queries. An improved method, named Pipeline Topology Method (LTM) has been developed, where the checking tasks are split into hierarchical groups that can be arranged as a pipeline. LTM has the benefit of reducing the total number of queries. However, this advantage comes at the cost of more communication load in the pipeline. To address this trade-off, we further proposed a performance model that balances the computation and communication costs and gives the best layout of the topology.

As the third contribution, we conducted a comparative study, based on the LUBM benchmark [40] using Twitter Heron [48]. Our experiments showed that the proposed solution LTM improves the system throughput of the baseline method NTM by up to 139%. We also demonstrated that the cost model could correctly optimize workload distribution by showing results in various metrics, such as the number of compiled operations, processing latency, and the memory cost.

6 Limitations

This section discusses some of the common limitations for the three projects in this dissertation. The specific limitations of each project can be found in the corresponding sections of Part II.

6.1 Individual Components vs. An Integrated System

Each research project of this thesis focuses on an individual component of an RSP system. However, complex applications may require an integrated system.

Studying each component separately allows us to understand the proposed techniques in-depth. Especially, it is fairly easy to disclose the impact of a tunable parameter on the system performance. For example, in the solution of RQ 1, we used a parameter to control the weight of the temporal recency and the past join history. By varying the value of this parameter, we can empirically reveal how it affects the result recall. When integrating different components together, the performance of the overall system depends on the interactions of many parameters. It becomes difficult to understand and optimize the system performance.

Given an individual component, we can usually design a deterministic performance model to guide the parameter tuning process. For example, our third project comes with a cost model that selects the best topology layout. However, after combining these components, the overall system performance cannot be accurately modeled anymore. The optimization goals in each method can even conflict each other, making the individual performance models obsolete. One possible solution is to apply black-box optimizations on the integrated system. Bayesian Optimization has shown its advantages in tuning parameters of complex systems [25, 32]. Therefore, when integrating all the components in

this thesis into one system, we can rely on some black-box optimization engines, such as Spearmint and Google Vizier to optimize its performance [38, 73].

6.2 Key Features vs. A Full-fledged System

Our studies only focused on some of the most important features in an RSP system. They still need to be generalized to become a full-fledged RSP system. Below, we list the key features in each project that need to be extended.

Methods in RQ 1 target on the join between two streams. Although it is one of the most important operators in an RSP engine. Other operations, like multi-way joins, projections, filters, and aggregation, should still be studied. In the paper, we briefly discussed how to adapt our method for these operations, but it still needs real implementations and performance comparisons.

The solutions of RQ 2 consider a complex query with two different sub-patterns: BGP and aggregation queries. Other RSP query components are not considered. For example, a FILTER clause can change the ranking method of data in the local view. Some recent efforts are extending the core idea of RQ2, such as [83, 84]. We believe there is still potential to further extend it, such as considering the OPTIONAL clause.

The reasoning procedure in RQ 3 only considers the DL-lite$_{core}$ logic. It needs to be generalized to other advanced Description Logics. One of the closest extensions is DL-lite$_{horn}$ [6], where parts of our algorithms need to be adjusted to cope with concept conjunctions. Another way to generalize our idea is to consider the $\mathcal{EL}^{++}$ logic [8, 9]. This logic involves transitive axioms that may cause loops in a topology. A possible solution is to manage these axioms as a separate procedure that can be executed before or after the topology.

7 Conclusions and Future Work

This thesis studies various methods to improve the efficiency of processing and reasoning of semantic streams. The three research projects address the challenges in the context of handling massive amounts of semantic streams. Each research project bears one of the **V**s as the primary target: the "Velocity" requirement is ensured by applying an eviction strategy to avoid system overloading; the "Variety" problem is addressed by an efficient maintenance process that accesses BGD with minimal cost; the "Volume" issue is resolved by performing scalable reasoning over significant amounts of streams to exclude implicit noise.

In the first project, we showed that the overloading problem occurred in two real-world datasets. The data-aware eviction strategy CLOCK addresses this issue by combing measures of recency with past performance history. It correctly retains the critical data and discards the less useful ones. Experimental results have shown that CLOCK together with its variants substantially outperforms the often-used LRU and FIFO strategies regarding result recall. As an extension of RQ 1, the next step is to investigate the eviction problem in a distributed environment, where operators of a system reside on different machines. When spikes appear in the workload of one operator, we need to decide which of its

upstream operators should perform eviction and how much data to evict. While our work is only a first step in investigating the overload problem, we believe it has a significant potential value of ensuring system reactiveness, especially on resource-limited systems.

In the second project, we showed that accessing remote BGD can be very expensive, waste resources, and deteriorate the response time. Refreshing a local view of BGD is necessary and subject to update-budget constraints. We propose different methods that efficiently allocate the budget to optimize the result freshness. Our solution relies on a bipartite graph to model the join between streams and BGD. It exploits the graph structure to improve response freshness for two kinds of sub-query patterns. We further proposed two extensions: the first one takes the future impact of each update into account. The second one flexibly allocates budget over different windows. Experiments have shown that our basic solution significantly improves the result freshness compared to the baseline algorithms (i.e., RAND and LRU). The two extensions further improve the performance of our basic solution. In future, we plan to extend our methods to other SPARQL operators as discussed in Section 6.2. We believe that our study highlights an important problem in RSP—the joint evaluation of stream and BGD under budget constraints—and provides solutions for different subqueries. As such, it paves the way for real-world RSP systems, where the integration of stream and BGD is ubiquitous.

The third project is inspired by the observation that streaming data are growing in schema complexity. We employ reasoning techniques over streams to detect inconsistencies. While this problem is usually studied in the context of static knowledge bases, we formally adapted it for the streaming setting and proposed scalable solutions that can be deployed on a DSPE. Our methods can compile a reasoning procedures into a processing workflow of a DSPE. The baseline method NTM adopts query-rewriting techniques to generate continuous queries that assess the consistency. To reduce the excessive number of queries produced by NTM, we proposed LTM, where the workload of consistency checking is distributed across a pipeline. We analyzed the trade-offs between computation and communication costs in LTM and introduced a cost model to optimize it. Experimental results suggest that LTM significantly outperforms NTM in terms of system throughput, based on the LUBM benchmark. In future, we will consider dynamically adjusting topology during run time. This is an important issue since the stream rate in real-world is usually changing rapidly, which requires a topology to be re-optimized from time to time. The stop-and-restart approach in the current systems incurs a significant down time. Therefore, reconfiguring a topology on-the-fly is an impending challenge to avoid this problem. Our methods of compiling consistency-checking topologies tackle one of the most difficult issues in stream reasoning–the system scalability. We believe that this study lays down the foundation for future scalable-reasoning research.

In conclusion, we witnessed the rise of semantic streams and related technologies in the past few years. We believe that this trend will likely continue. Processing and reasoning of semantic streams will have a wider range of application areas. We hope that this study could contribute to the research of semantic streams in future and give insights for designing new stream-related systems.

Part II

Contributions of this thesis

The CLOCK Data-Aware Eviction Approach:
Towards Processing Linked Data Streams with Limited Resources

This chapter is based on a submission that was accepted at the:

Proceedings of the 14th Extended Semantic Web Conference
pages 6–20, 2014

The CLOCK Data-Aware Eviction Approach: Towards Processing Linked Data Streams with Limited Resources

Shen Gao, Thomas Scharrenbach, and Abraham Bernstein

Department of Informatics, University of Zurich, Switzerland

Abstract. Processing streams rather than static files of Linked Data has gained increasing importance in the web of data. When processing data-streams, system builders are faced with the conundrum of guaranteeing a constant maximum response time with limited resources and, possibly, no prior information on the data arrival frequency. One approach to address this issue is to delete data from a cache during processing – a process we call *eviction*. The goal of this paper is to show that data-driven eviction outperforms today's dominant data-agnostic approaches such as first-in-first-out or random deletion.
Specifically, we first introduce a method called CLOCK that evicts data from a join cache based on the likelihood estimate of contributing to a join in the future. Second, using the well-established SR-Bench benchmark as well as a data set from the IPTV domain, we show that CLOCK outperforms data-agnostic approaches indicating its usefulness for resource-limited linked data stream processing.

1 Introduction

Streams of Data have become increasingly common in the Web of Data (WoD). Constant streams of weather data, stock ticker information, tweets, bids on an auction site, and TV viewers switching channels are all examples of such streams. When processing such streams, one typically attempts to answer queries or evaluate some functions as data comes along. To that end, SPARQL-like [41] languages such as SPARQLStream [18], C-SPARQL [12], CQELS [51], TEF-SPARQL [46], and EP-SPARQL [3] were proposed to allow joining elements of the stream with each other or some rich background data set.

In contrast to static data processing systems, *stream processing systems need to be reactive*: they must process continuously arriving new data within a given set of Quality of Service (QoS) constraints. Given that latency (or the delay by which newly incoming data impacts results) is usually among these constraints, Little's law [55] 'commands' that we change from all-time semantics to one-time-semantics: data arriving after the accepted latency will not influence an answer produced by the system. Consequently, stream processing systems have to implement measures to cope with situations where the incoming data-rate overwhelms the systems' processing capabilities – a situation we call a *stressed* system. Stress, in turn, occurs either because the constant data rate is overwhelming, hence the environment is *overloaded*, or *bursts* in the data-rate inundate the system.

Current systems typically try to avoid stress by limiting the scope of the query using a time-window – a language feature many systems support to define the *context* of a query. This solution is, however, limited to situations in which the window that is semantically relevant according to the application domain limits the arriving data to volumes that can

be handled by the system. Hence, even in the light of query contexts, it is easy to imagine a use case with a data rate that will overwhelm the system.

In order to deal with stress, stream processing systems can sample the incoming data, an operation called *load shedding* [10, 24, 77]. In this paper, we propose *to delete data from the caches of the operators*, as this operation can exploit the state of the operators in addition to data statistics to reduce stress. We refer to this as *eviction*, as it expels data items from the cache of operators. Both load-shedding and eviction allow maintaining the QoS constraints of a stream processing system in the light of limited resources. They do so at the cost of possibly introducing *errors*: mistakenly evicting data-items from intermediate caches that would lead to results can lower recall and even precision (when using the 'non-open world assumption' operators such as average).

This paper proposes the computationally efficient data-aware eviction strategy Clock that evicts data from a join cache based on an estimate of contributing to a join in the future. Specifically, we show that our method outperforms data-agnostic strategies such as random or First-In-First-Out (FIFO) using both SRBench, a standard benchmark for evaluating the performance of Linked Data stream processing systems [85], and a real-world IPTV data set. As such, the paper extends a preliminary study that showed that an omniscient eviction strategy (i.e., a strategy that could look into the future) could outperform data-agnostic scheduling strategies [63] and makes it practical due to the removal of the reliance on future knowledge.

Consequently, we address the following Research Questions (RQ):

RQ 1: Real-world datasets, such as the ones in our study, can induce stress even when context limitations are present.

RQ 2: Eviction can curb memory consumption at the cost of a lower recall.

RQ 3: Our Clock data-aware eviction strategy outperforms data-agnostic eviction strategies in terms of recall.

RQ 4: Clock outperforms the Least Recently Used (LRU) strategy, which is often used in cache management, in terms of recall.

Outline: After a conceptualization of load shedding and eviction for processing streams of data (cf. Section 2), Section 3 presents our Clock method, followed by a thorough evaluation of our research questions on two real-world data sets (cf. Section 4). After a discussion of limitations (cf. Section 5) and related work (cf. Section 2) we close with a summary of our findings (cf. Section 7).

2 System Model: A Conceptualization of Load Shedding and Eviction

A data stream processing system can be conceptualized as Processor P that continuously consumes one or more input data streams IS_i and transforms them through a series of operators O into one or more internal flows IF_j, some of which are emitted as output data streams OS_j. Hence, $P = (IS \cup OS \cup IF, O)$ can be seen as a directed graph, where the data flows along directional edges $(IS \cup OS \cup IF)$ that connect the operators $o_i \in O$,

which are the nodes. All internal flows $if \in IF$ connect two operators, whilst the input streams IS_i and output streams OS_j are only connected to one operator.

In the context of the WoD the input streams IS_i typically consist of sequentially arriving data tuples of the format $< s, p, o > [t_{start}, t_{end}]$, where $< s, p, o >$ is a triple representing a fact and t_{start} / t_{end} denotes the start/end time of the triple's validity. Alternatively, when $t_{start} = t_{end}$ (i.e., the triple describes an event at time t rather than a fact) the incoming tuples can be abbreviated as $< s, p, o > t$ or, when only relative temporal order is implied by arrival time, t can be dropped. The output streams OS_j contain a continuous sequence of tuples either in the same format as the ones in the input stream or denoting bindings to a query. Note that all our considerations do not take the format of the input and output into account. Hence, our findings generalize to all stream processing systems.

System Stress This conceptualization indicates that a system can be stressed either by overwhelming the load on the operators or by inundating the bandwidth and latency constraints on the edges. This paper will focus uniquely on the former problem: It will assume that the bandwidth/latency constraints of the edges are adequate for tasks at hand. Note that operators can be overwhelmed either by time complexity (e.g., an operator that computes the factorial of large numbers) or by space complexity (e.g., a join that has to maintain a cache).

A context can curtail stress, as it allows the system ignoring nonsensical data-items and concentrating on data relevant for answering a query. A context is defined for an operator and defines which data is valid for evaluating the operator. One oftentimes used context is a time-window. Consider we want to count the audience for a certain TV channel based on a stream of events indicating which viewer switches to what TV channel. We need to know the set of data items the count is based on, i.e., the context of the operation. Prudent choices are, for example, time-based windows such as the last second (referring to the current TV ratings) or the past hour (referring to past ratings). For a detailed overview over windows and operators, we refer to [22]

Dealing with Stress We know of two approaches for dealing with stress: load shedding and eviction.

In *load shedding* the stream processing system samples the input streams and only considers part of the data. Formally, it is a sample operation $s : IS_i \mapsto \widehat{IS_i}$, where $\widehat{IS_i} \subset IS_i$. Figure 1 illustrates this for a join between stream IS_x and stream IS_y. Here stream IS_x sheds its data item x^5 at $t = 3$ by deleting it from the considered input stream. Load shedding strategies range from deleting data at random (e.g., useful for dealing with high-frequency sensor reporting averages per time unit), via a scheduling strategy such as FIFO, to estimating statistics of which data to delete and which not [10, 24, 77].

In *eviction* the stream processing system removes data from the internal memory of the operators to preserve computational resources. Formally, eviction is the extension of the operators $o_i \in O$ with one or more eviction strategies $es : memory \mapsto \overline{memory}$, where $\overline{memory} \subset memory$. Figure 1 illustrates two eviction strategies: First, it 'garbage collects' items that exit the context windows win_{now} of streams IS_x and IS_y. At time $t = 2$ for both streams these are all data items, which we observed at $t = 1$, i.e., x^2 and y^2. Second,

due to the limited size of its join cache, it decides to remove data item y^3 of stream IS_y. Note that this second strategy removes a data item which we observed at $t = 3$, i.e., a data item which would be still valid concerning the context of stream IS_y.

This paper focuses on the impact of eviction strategies on the potential error in the resulting data. Specifically, the next section will introduce two traditional, data-agnostic evictions strategies (e.g., random eviction and FIFO), one based on the nature of the data (e.g., garbage collection) as well as our CLOCK strategy, which relies on the likelihood of future joins.

3 Eviction Strategies

Eviction removes items from the internal memory (or cache) of an operator to save space. Most WoD stream processing systems extend the SPARQL algebra in order to allow evaluation of SPARQL operators on streams. As a consequence, the operators' caches typically hold candidate variable bindings. Hence, the role of the eviction strategy is to choose variable bindings to delete from the cache.

Formally, an operator's cache (or short cache) C_{op} with *limit* M and *size* N is a finite set of variable bindings $\mu_1, \ldots, \mu_N$, where $N \leq M$. We say there exists an overflow for C, if and only if $N > M$, i.e., in case the number of items in the cache exceeds the cache's

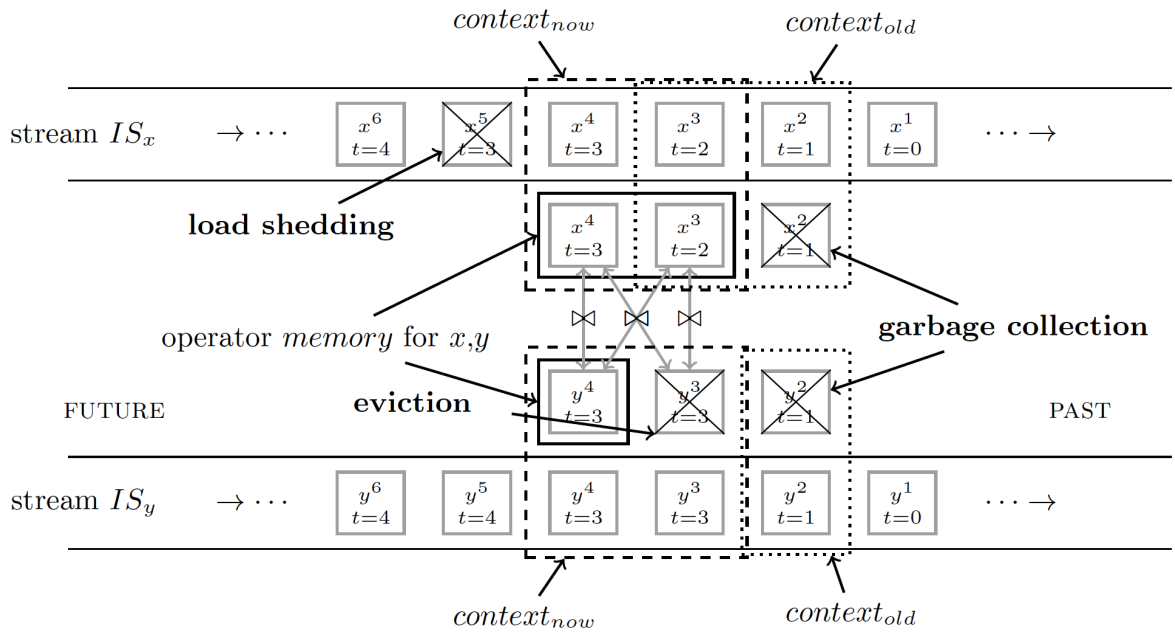


Fig. 1: Depiction of stress handling approaches in a join of two input streams. Load shedding on input stream IS_y, garbage collection on both join caches, and other (unspecified) eviction on join cache of stream IS_x. Context is shown as windows (dashed: now, dotted: past) and cache memory sizes are two items for the upper and one item for the lower stream.

limit. An eviction strategy es removes data items from a cache C such that $C' = es(C, M)$ is a cache of limit M and $|C'| \leq M$, i.e., it has no overflow. In the sequel, we define different eviction strategies.

Note that in this study we consider eviction for caches of two-way-joins, i.e., joins with two join partners sharing one common join variable. We discuss possibilities for extensions to other operators in Section 5.

3.1 Baseline Eviction Strategies

In this section, we succinctly introduce the four baseline or traditional eviction strategies: random, FIFO, LRU, and garbage collection.

Random eviction deletes variable bindings from a cache according to a uniform distribution $U(0, N)$ over all cache entries. To deal with cache overflow, it requires to compute $O(N - M)$ random indices to delete from the cache.

First-In-First-Out (FIFO) maintains a queue of items, where the head of the queue is deleted whenever an overflow occurs. It requires $O(N - M)$ calls to the queue. Together with random eviction FIFO has been adopted by today's conventional systems [77].

Least-Recently-Used (LRU), a strategy widely adopted in cache management including the SASE+ stream management system [31], extends FIFO by moving items to the back of the deletion queue whenever they are accessed. As with FIFO, handling an overflow requires $O(N - M)$ operations on the LRU queue.

Garbage Collection removes irrelevant data items from the operator cache. Relevancy may be determined via the context of a query. When processing TV viewership data, e.g., current viewers of a program are determined by joining the most recent program changes and user channel switches. Older channel switches by a user can, therefore, safely be garbage collected, as they are irrelevant to the query. Pure garbage collection is an incomplete eviction strategy, as it may not be able to remove enough items from the cache, when the context is not sufficiently restrictive.

Following the example of Section 2, random eviction would delete user sessions at random while FIFO would delete the oldest sessions – both while the session would be still valid. In a data agnostic way they 'blindly' follow their eviction strategies independent of possible future results. Garbage collection would delete all invalid sessions. It relies on data context but ignores the performance of the item in contributing to the operation. As a metric of past performance LRU deletes valid sessions with no recent activity. It favors temporal recency but ignores the magnitude of a binding's past performance. In the next subsection we introduce our CLOCK approach that estimates the future likelihood of usefulness based on past performance. It extends LRU by considering both recency and magnitude of usefulness.

3.2 The Clock Strategy

Clock is a data-aware eviction strategy that considers both recency and magnitude of past usefulness of a binding to estimate the likelihood of future usefulness, which it employs as criteria for eviction. Clock associates each binding with a score. Whenever an item is matched, it increases that score. When it looks for items to evict, it first depreciates the bindings' scores and then evicts those with lower scores. Thus, the score combines a measure of recency with a measure of magnitude.

Specifically, Clock maintains a circular buffer cache of M slots containing the bindings μ with their associated scores w_μ and a pointer to a position p in the circle.[1] When a new data item arrives, it gets assigned an initial score $w_\mu = w^0$. If there are empty slots, it is added to one. Else the pointer depreciates the score of the item at position p using the depreciation function $dep()$. If the item's new score is lower than some threshold τ, then it gets evicted, and the newly arrived binding takes its place. Otherwise, the pointer moves to the next position and repeats this procedure. Whenever a binding contributes to a join, its score gets increased by one (i.e., $w_\mu := w_\mu + 1$).

Following the example of Section 2, Clock increases the count whenever we observe a session activity, i.e., a user switches channels. At each point in time we decrease the count whenever we observed no activity.

Practically, we propose two different depreciation functions. The linear depreciation function $dep_{lin}(w) = w - 1$ just decreases the value of a score by one. It is associated with the threshold $\tau = 0$. Alternatively, we can depreciate exponentially with a depreciation rate ρ resulting in $dep_{exp}(w) = w * \rho$ $(0 < \rho \leq 1)$. In this case w_μ will never reach 0. Hence, we picked $\tau = 0.01$ as a threshold. We call this extended version Clock$_{exp}$.

In its baseline description without any extensions, Clock may have to circle around the cache a number of times before finding a suitable candidate for eviction. With an extension containing the currently smallest score in the cache Clock needs at most $O(M)$ (limit of the cache) depreciation steps to find a victim for eviction. Clock also requires a constant amount of additional memory (in particular M) for storing the scores w_μ of the bindings.

Observations: First, as mentioned, Clock can be seen as an extension of LRU that considers both temporal recency and past join history. The weight between these two factors can by set by adjusting ρ.

Second, the initial score w^0 reflects the degree to which we give a binding μ an initial chance to find a join partner. It should be sufficiently high, such that it has a chance to survive initially. It should be sufficiently low to ensure the timely eviction of less useful bindings. In Clock$_{exp}$ it determines together with ρ how dynamic the eviction strategy is.

Third, Clock could be easily extended to multi-way joins by using different-sized increments for partial vs. full join results.

Fourth, the Clock eviction strategy is founded on the following assumptions: in burst streams, eviction only takes place eventually. As a result, cache entries for which we

[1] Using a circular cache allows us to efficiently find eviction candidates by circular iterations over the buffer.

observed no join partners could remain in the cache for a long time until eviction takes place. In an overloaded environment, there is only little chance that such items stay in the cache for long periods.

4 Evaluation

This paper argues that real-world and, hence, resource-limited WoD stream processing systems will be subject to stress even when using a use-case motivated context to limit the data that needs to be taken into consideration. To deal with stress, it proposes to employ eviction – an approach that removes data from the caches of the operators of the stream processor. Specifically, it suggests to employ a data-aware eviction strategy over (more traditional) data-agnostic eviction strategies and introduces the CLOCK approach that is based on a likelihood estimate of future usefulness of an item.

To support this argumentation, this section will provide empirical evidence for the research questions (RQ) we defined in Section 1:

RQ 1: Real-world datasets, such as the ones in our study, can induce stress even when context limitations are present.
RQ 2: Eviction can curb memory consumption at the cost of a lower recall.
RQ 3: Our CLOCK data-aware eviction strategy outperforms data-agnostic eviction strategies in terms of recall.
RQ 4: CLOCK outperforms the Least Recently Used (LRU) strategy, which is often used in cache management, in terms of recall.

As a consequence, this section will first lay out the experimental setup (Section 4.1) and then proceed to discuss each of these research questions in turn. We first show that our data sets can be used to evaluate RQ2 and RQ3 (Section 4.2). We then evaluate these with two different experiments: first, we show the general performance of CLOCK versus other strategies (Section 4.3), then we show that we can optimize CLOCK with regards to learning its parameters (Section 4.4).

4.1 Evaluation Setup

To evaluate our research questions we built a *stream processing simulator* that allows to precisely measure, curb, and manipulate the memory consumption of the involved operators via pluggable load shedding and eviction strategies. Whilst the system does correctly identify the bindings, we call the system a simulator rather than a full-fledged stream processing systems as it was built for experimentation rather than efficient processing and lacks elements such as a query parser/optimizer.

Given our research questions, the *Key Performance Indicator (KPI) of our evaluation is recall*, which is defined as the ratio between the number of results with a given cache size to that with unlimited cache size. We disregarded the time complexity of the eviction strategy as we found that all the strategies were faster than 40 ms ($\mu = 9.45ms, var = 15.03ms$) per data item – a performance we deem sufficient for most applications.

To ensure realistic data, we employ *two real-world data sets*: SRBench and ViSTA-TV. *SRBench* [85] is a well-established benchmark for assessing the semantic streaming processing engines. It comprises the LinkedSensorData, GeoNames and DBpedia.[2] Our test query focuses on the wind speed data set because it is reported by most of the sensor stations. To simplify our experiments, we preprocessed the SRBench dataset and extracted all of the 603'642 windspeed data entries, where each triple has the format: $< sensorID, \mathsf{reports}, windSpeed > time$. Since the queries of *SRBench* were designed to benchmark the functionality of different engines, we designed a new query focused on establishing the performance of eviction strategies. The query (cf. Listing 1), defined using the TEF-SPARQL [46] semantics, aims to find sensors with similar wind speeds using a self-join on the *windSpeed* entry – an operation, where recall depends greatly on the size of join-cache employed.

```
SELECT ?sensor1, ?sensor2 FROM STREAM windSpeed
WHERE {
     ?sensor1  reports  ?windSpeed ?T1 .
     ?sensor2  reports  ?windSpeed ?T2 .
     FILTER (?sensor1 != ?sensor2) .
     FILTER(?windSpeed >= 10^^xsd:int) .
}
CONTEXT((?T1 - ?T2) <= 200^^xsd:millisecond) .
```

Listing 1: A self-join query inspired by SRBench

ViSTA-TV[3] is a FP7 financed EU project that investigates the real-time processing of TV viewership information. The data set we employed for evaluation contains anonymous IPTV viewership logs (Log) in the format $< userID, \mathsf{watches}, channelID > [t_{start_{viewer}}, t_{end_{viewer}}]$ and Electronic Program Guide (EPG) data $< channelID, \mathsf{plays}, programID > [t_{start_{EPG}}, t_{end_{EPG}}]$. Each data entry is annotated by a starting time stamp and an ending time stamp. A data entry is consider to be expired when the system time has passed its ending time. We used three-day's Log and EPG data, which contains 1'887'256 viewership events and 31'960 EPG entries. As defined in TEF-SPARQL [46], the query (cf. Listing 2) is a two-way join operation, which represents the use case to find all users that are currently watching a specific TV-program. To ensure that all caches were in a steady state, first one-third amount of data in each data set are used to 'warm up' the system and the rest are reported here.

All experiments were conducted on a MacBookPro with a 2.7 GHz Intel Core i7, 16GB of RAM, and 256 GB of SSD disk space running Mac OX 10.9.1.

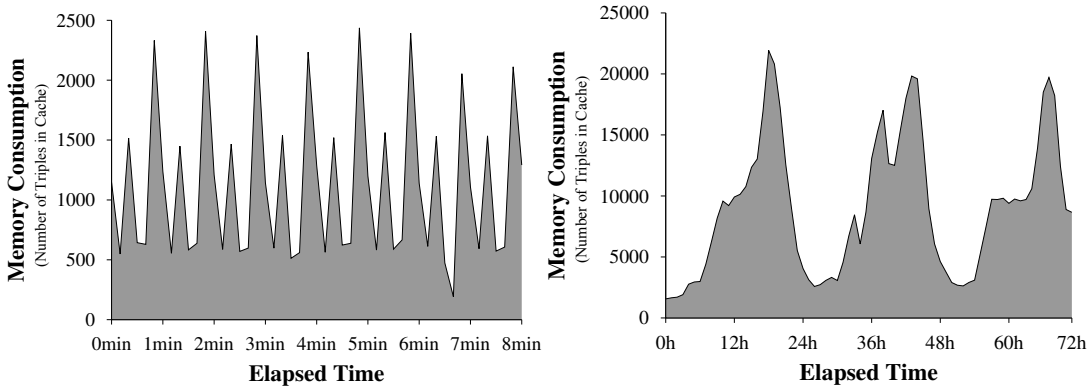
[2] http://wiki.knoesis.org/index.php/LinkedSensorData, http://geonames.org, http://dbpedia.org
[3] http://vista-tv.eu/

```
SLECT ?user, ?program FROM STREAM Log, EPG
WHERE{
     ?userID    watches ?channelID   ?T_start_viewer ?T_end_viewer .
     ?channelID plays   ?programID   ?T_start_EPG   ?T_end_EPG .
}
CONTEXT((!?T_end_viewer < ?T_start_EPG) && (!?T_end_EPG < ?T_start_viewer)).
```

Listing 2: ViSTA-TV query

4.2 RQ1: Real-world Systems are Subject to Stress

To elucidate if real-world systems are likely to be subject to stress, we graphed the cache sizes necessary to fully answer our queries for the two datasets. In other words, we assumed a system without any memory limitations and elaborated how much memory (i.e., number of triples inside cache) it needed to provide correct answers (i.e., 100% precision and recall) to our queries. Figure 2a/2b graphs 8 minutes/72 hours worth of data measured every 10 seconds/1 hour for SRBench/ViSTA-TV.

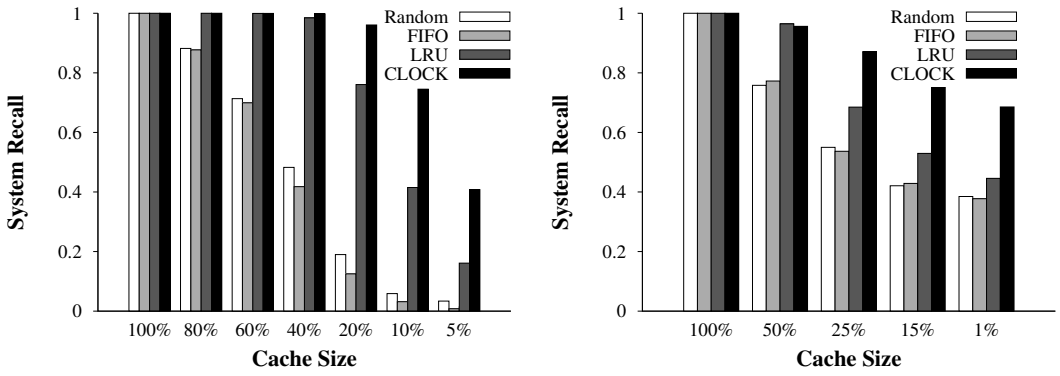


(a) SRBench **(b)** ViSTA-TV

Fig. 2: Fluctuations in memory consumption per time unit.

We can observe significant fluctuations in the memory size needed irrespective of the context limitations provided by the queries (e.g., the limitation on a $200ms$ window in the SRBench case). In SRBench, this is because some sensors cluster their reporting. In ViSTA-TV, the start/end times of major shows may lead to fluctuations in load.

Whilst these findings do not provide proof that systems will undergo stress conditions, they strongly indicate that real-world systems are subject to massive changes in load (hence stress). Consequently, we can argue that for any real-world system there would be a real-world dataset that would overwhelm the available resources either by overloading or by bursts. This, in turn, would argue for systems that are resilient against stress supporting the premise of this paper and answering *RQ1*.



(a) SRBench **(b)** ViSTA-TV

Fig. 3: System recall with varying cache size.

4.3 RQ2-4 Eviction Results: Memory Consumption and Recall

The fact that eviction can curb memory consumption is almost self-evident. Obviously, randomly deleting data items whenever a cache-size limit is met will curb cache size. The more interesting question is what the cost of the memory limitations would be in terms of recall for a given eviction strategy.

We measured the recall gained with different cache sizes for four eviction strategies: Random, FIFO, LRU, as well as CLOCK using the linear depreciation function $dep_{lin}()$ with $\tau = 0$. Note that we did not include our prior approach [63], as it can only be used offline due to its reliance on the whole dataset; including items not yet encountered in the stream.

The results are reported in Figures 3a and 3b. All strategies were combined with garbage collection to give them the advantage of logically evicting data items that would not be used anymore.[4] We can make the following observations:

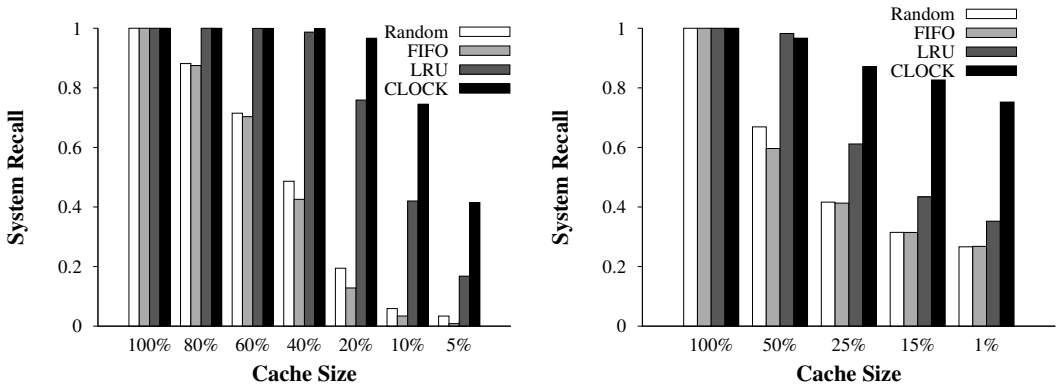
First, all strategies perform similarly with large cache sizes: systems with sufficient memory are unlikely to be stressed. Hence, eviction does not impact recall significantly.

Second, with decreasing cache size, the data-aware strategies strongly outperform Random and FIFO by up to 78% and 81% in ViSTA-TV and 12 and 50 times in SRBench. These results show that a *stressed* system with limited memory resources dramatically benefits from data-aware eviction strategies.

Refinement under Stress To further highlight these results, Figures 4a and 4b plot the performance results under *stressed* conditions. Hence, recalls are computed only during the number of data items per second surpassed the respective average input rate of SRBench and ViSTA-TV. The results further reinforce the above findings: CLOCK outperforms the traditionally employed LRU by up to 147% for SRbench and 162% for ViSTA-TV.

These results provide evidence to answer *RQ2*, *RQ3*, and *RQ4*. We can clearly conclude that for the given data sets data-aware methods outperform data-agnostic methods in the

[4] Note that we cannot measure garbage collection alone, as it does not guarantee limited cache size usage.



(a) SRBench **(b)** ViSTA-TV

Fig. 4: Results of a *stressed* system.

light of resource constraint. Further, we established that our CLOCK strategy outperforms the traditional LRU approach. What remains open is how robust CLOCK is towards varying depreciation functions. Specifically, how does CLOCK compare to CLOCK$_{exp}$ with different depreciation weights ρ that we discussed in Section 3.2 – a topic we will investigate in the next subsection.

4.4 Tuning CLOCK via Varying Depreciation Weights ρ

Different data sets may exhibit varying degrees of 'decay' in the applicability of their data items. We, hence, investigated if CLOCK could be better tuned to a data set using the depreciation functions *dep*. Specifically, we ran both CLOCK and CLOCK$_{exp}$ on our two data sets. For CLOCK$_{exp}$ we varied ρ between the following values: $\rho \in \{0.95, 0.57, 0.5.0.25\}$.

Figure 5 shows heat-maps depicting the recall for both SRBench (on the left) and ViSTA-TV (on the right). The heat-maps clearly show that the depreciation rate ρ has a profound influence in recall. For example, in SRBench, the best performance is obtained when $\rho = 0.95$. In the ViSTA-TV data set, $\rho = 0.5$ seems to provide the best performance for smaller cache sizes. Hence, in the ViSTA-TV data set it appears to better emphasize more on recent items and depreciate results faster than in SRBench. Consequently, CLOCK can be tuned according to the idiosyncrasies of a data set by choosing an appropriate depreciation rate. We hope to investigate automated tuning in the future.

5 Limitations

First, our current evaluation is limited to one operator: the join. We believe that focusing on joins for a first study made sense, as it is both the most used operator and one of the most intricate. As mentioned in Section 3, our CLOCK method could be easily extended to multi-way joins. Projections can be supported without any cache. Aggregation functions have constant memory implementations or approximations requiring investigations

	SRBench					ViSTA-TV			
	60%	*40%*	*20%*	*10%*	*5%*	*50%*	*25%*	*15%*	*1%*
LRU	0.972	0.919	0.786	0.571	0.266	0.964	0.635	0.529	0.446
CLOCK_exp 0.25	0.991	0.966	0.875	0.654	0.291	0.964	0.865	0.740	0.526
CLOCK_exp 0.5	0.998	0.989	0.936	0.770	0.413	0.932	0.818	0.762	0.683
CLOCK_exp 0.75	0.999	0.997	0.965	0.852	0.545	0.965	0.789	0.753	0.683
CLOCK_exp 0.95	0.999	0.998	0.979	0.896	0.650	0.961	0.761	0.694	0.622
CLOCK	0.997	0.992	0.967	0.875	0.567	0.956	0.799	0.751	0.685

Fig. 5: Parameter tuning for CLOCK and CLOCK$_{exp}$ (with $\rho \in \{0.95, 0.57, 0.5.0.25\}$) on SRBench and ViSTA-TV.

similar to ours. Filters are interesting, as their implementation will greatly depend on the definition of context.

Second, not neglecting the importance of throughput and latency, we deliberately focused on the very KPI that eviction will impact negatively, i.e., recall. Other metrics will be evaluated when we implement CLOCK in real stream processing systems. Despite this limitation we believe that CLOCK's performance regarding throughput is comparable with other methods, given its low computational overhead (cf. Section 3).

A disadvantage of CLOCK is that it has to invest additional memory for storing the scores w_μ. With the same amount of memory, methods like FIFO and LRU may, hence, cache more bindings than CLOCK. However, this overhead could be minimized by implementing the score as a bitmap. Moreover, as CLOCK only needs to adjust the score for each binding, its implementation is orthogonal to other internal memory structures (e.g., a B-tree) and will not impose extra overhead on them. A next study will have to investigate the trade-off between using some memory for eviction-bookkeeping and using it only for storing bindings.

Last but not least, we will need to consider additional datasets. Whilst the two data sets considered come from two vastly different real-world applications we believe that much more data characteristics. The compilation of more good data sets for WoD stream processing seems to be a challenge for the whole community.

6 Related Work

We discuss related work in the followings. We will first introduce different Semantic Flow Processing (SEP) systems and then discuss query processing in memory-constrained environments. Finally, we review related load shedding strategies for data stream processing.

Semantic Flow Processing Systems C-SPARQL [12] performs query matching on subsets of the information flow, which are defined by windows. The decidability of SPARQL query processing on such windows of RDF triples causes the number of variable bindings produced to be finite. However, the size of variable bindings may still become prohibitively

large, e.g., when using non-shrinking semantics for aggregates [13]. For a cache of a given window size, our eviction strategies could be directly applied.

EP-SPARQL [3] and TEF-SPARQL [46] are both complex event processing systems for semantic data flows. EP-SPARQL extends the ETALIS system with a flow-ready extension of SPARQL. TEF-SPARQL distinguishes between *Events* that happen at a specific time point and *Facts* that remain valid until some events alter them. Both systems incorporate a garbage collection facility that can "prune outdated events". Since garbage collection is orthogonal to our strategies (cf Section 4.3), our findings are directly applicable to these systems.

CQELS [51] "implements the required query operators natively to avoid the overhead and limitations of closed system regimes". It optimizes the execution by dynamically reordering operators because "the earlier we prune the triples that will not make it to the final output, the better, since operators will then process fewer triples". This pruning does, however, not make any guarantees about the number of variable bindings created by the processors. Our methods should be directly applicable to CQELS as it provides a native implementation of the operators which contain lists of active variable bindings.

Query Processing in Memory-Constrained Environments In memory-constrained environments various techniques have been proposed to reduce the memory footprint of query planners and the number of intermediate results.

Targeting SPARQL queries Stocker et al.[74] investigated the selectivity estimates to optimize query execution. To efficiently generate alternative query plans, [17] proposed a branch-and-bound to enumerate join plans for left-deep processing trees. This method requires less memory as it prunes the search space during enumeration. Our eviction strategies are designed for caches and assume a given query execution plan.

Regarding multiple aggregate queries over stream data Naindu et al. [62] proposed a new hash model for estimating the cost for intermediate aggregates. This method groups common attributes of related queries and reduces overall memory usage. Based on this new model, they also proposed a greedy heuristic to generate the execution plan. Our eviction strategies are designed for general semantic streaming systems that perform not only aggregate query, but also other kinds of queries.

In a XML processing system the memory consumption for XML processing can greatly exceed the actual file size. Therefore, an entire XML document may not fit into main memory. In [57] the authors proposed a method that analyses XQuery to identify and extract only useful attributes form XML documents during compilation to reduce the file size. Our eviction strategies deal with semantic data, where it is straightforward to identify useful attributes from input stream. Meanwhile our strategies are also applicable to projected variable bindings.

Load Shedding Load shedding has been applied to information flow processing. Approaches like [10, 24, 77] perform load shedding by dropping tuples from the stream, i.e., dropping data instead of variable bindings.

In [77] the authors proposed to insert a "drop operator" into the query execution plan, which automatically decides where, when and how to perform load shedding. Regarding

how to perform load shedding they proposed a random method as a baseline and a "semantic method" which decides whether to retain a data entry based on estimating its impact on QoS. In addition to their approach, our strategies also take into account the time a data entry has resided in memory. Similar to [77], [10] also proposed a special operator that decides where and when to drop unprocessed data by using statistical methods. However, [10] only focuses on aggregate queries.

SASE+ [31] employs an automata-based matching approach. Similar to our case of caching variable bindings, SASE+ stores automata states. The authors do apply some eviction strategy. However, their strategy is based on a deterministic approach that is similar to FIFO and LRU in our baseline approaches.

Finally, Das et al. [24] propose a simple equi-join on two incoming streams and to evict tuples that are unlikely to find a join partner. However, this method works only with a sliding window and with a single equi-join of two streams. Our approaches could be applied on caches for any kind of join.

7 Conclusion and Outlook

In this paper, we presented our data-aware eviction strategy CLOCK, which addresses stress in WoD stream processing systems. We found that stress in terms of overloading and bursts occurred in our two real-world datasets. In addition, CLOCK and its variant CLOCK$_{exp}$ outperform the often-used LRU strategy by factors between 1.5 and almost 3 and FIFO strategy by even higher factors.

The next step in our investigation will be to implement these strategies in a real stream processing system to study the trade-off between recall and other KPIs such as latency and throughput with different data sets. Whilst our work is only a first step in investigating resource-limited stream processing, we believe it pursues an important direction that sets the expectation for the real-world usage of such systems.

Acknowledgements

We would like to thank Khoa Nguyen for all his earlier work on this topic and Daniel Spicar for his constructive advice. The research leading to these results has received funding from the European Union Seventh Framework Program FP7/2007-2011 under grant agreement No.296126.

Planning Ahead: Stream-Driven Linked-Data Access under Update-Budget Constraints

This chapter is based on a submission that was accepted at the:

Proceedings of the 15th International Semantic Web Conference
pages 252–270, 2016

Planning Ahead: Stream-Driven Linked-Data Access under Update-Budget Constraints

Shen Gao[1], Daniele Dell'Aglio[2], Soheila Dehghanzadeh[3], Abraham Bernstein[1], Emanuele Della Valle[2], Alessandra Mileo[3]

[1] Department of Informatics, University of Zurich, Switzerland
[2] DEIB, Politecnico di Milano, Italy
[3] INSIGHT Research Center, NUI Galway, Ireland

Abstract. Data stream applications are becoming increasingly popular on the web. In these applications, one query pattern is especially prominent: a join between a continuous data stream and some background data (BGD). Oftentimes, the target BGD is large, maintained externally, changing slowly, and costly to query (both in terms of time and money). Hence, practical applications usually maintain a local (cached) view of the relevant BGD. Given that these caches are not updated as the original BGD, they should be refreshed under realistic budget constraints (in terms of latency, computation time, and possibly financial cost) to avoid stale data leading to wrong answers. This paper proposes to model the join between streams and the BGD as a bipartite graph. By exploiting the graph structure, we keep the quality of results good enough without refreshing the entire cache for each evaluation. We also introduce two extensions to this method: first, we consider a continuous join between recent portions of a data stream and some BGD to focus on updates that have the longest effect. Second, we consider the future impact of a query to the BGD by proposing to delay some updates to provide fresher answers in future. By extending an existing stream processor with the proposed policies, we empirically show that we can improve result freshness by 93% over baseline algorithms such as Random Selection or Least Recently Updated.

1 Introduction

Real-time processing of massive, dynamically generated stream-data has become increasingly popular on the Web [56]. In stream processing, one common task is to enrich the streams with external background data (BGD). This kind of tasks has to deal with two **V**'s of "Big Data" at the same time: *Velocity*, the rapidly changing nature of the stream data; *Variety*, integrating data from different sources[4]. RDF Stream Processing (RSP) has provided necessary languages to declare this task. Current RSP languages, such as C-SPARQL [14], SPARQL$_{stream}$ [19], and CQELS-QL [51], support complex queries that involve both streams and remote BGD. However, these RSP engines are not optimized for remote BGD access. Usually, they continuously fetch BGD to match newly arrived stream data ignoring the communication and potential financial cost of such operations. To improve BGD access, RSP engines may adopt local views (or caches), as done in database systems [39]. However, the remote BGD is not always static. Indeed, even in the mostly static linked-data realm, information changes [45]. Hence, the *freshness* of local views in the RSP engine degrades over time as updates in BGD do not propagate to the local view. To address this problem, RSP engines have to *maintain* the local view, by identifying the out-of-date (or *stale*) data items and replacing them with the up-to-date (or *fresh*) values

[4] http://www.ibmbigdatahub.com/infographic/four-vs-big-data

retrieved from the remote. Examples of such updating behavior include the identification of opinion makers in social media based on a stream of posts and (slowly-changing) contact-networks as BGD or traffic prediction based on position data fetched from mobile phones.

Maintaining a local view can take time. Given that a federated query evaluation can spend up to 95% of its time on accessing remote data [60], query evaluation under response time constraints becomes a major challenge. To ensure a certain response time, only a limited number of remote accesses can be allowed. Additionally, BGD providers may impose constraints such as API rate limits, e.g., Twitter[5]. Lastly, other communication and financial constraints may have to be considered, since accessing BGD can cost money, computation power or energy (in both the RSP engine and the remote service). Returning to the above examples, computing updated network metrics for opinion makers is computationally expensive, and fetching location updates from cell phones burdens scarce battery power. In this paper, we consider these constraints as a limited *budget* that restricts the number of possible BGD accesses. We study the problem of how to utilize the limited budget so that it can provide fresher response to the query.

To optimally manage BGD accesses under realistic budget constraints, this paper proposes to allocate budget only to carefully selected "important" data in the local view. Our algorithms exploit characteristics of the join between the stream and the BGD to improve the response freshness. Specifically, our contribution is threefold. First, we propose an algorithm that employs a bipartite graph to model the join selectivity between stream and BGD. It favors the update of data items with a higher selectivity within a budget constraint. This problem decomposes to two scenarios: one can be tackled with a local optimal approach; a second is NP-hard requiring a greedy heuristic approach. This encodes Hypothesis **H1: A maintenance processes exploiting join selectivity improves response freshness**. Second, we extend the above model to favor data items that have a longer impact on the response freshness, which leads to hypothesis **H2: Leveraging the definition of the sliding window and BGD change frequencies improves response freshness**. Third, we explore the trade-off between the current and future importance of data elements. We present an algorithm that exploits the change frequencies, join selectivity, and the sliding window all together to delay some current refreshes in favor of future, more important ones. It encodes hypothesis **H3: Considering both current and future evaluations for budget allocation further improves response freshness**.

Outline: Section 2 introduces some background of RSP and BGD access. Section 2 reviews related work. Section 4 formalizes the problem. Our solutions and their optimization are in Section 5. Section 6 provides evaluation results of our hypotheses on both real and synthetic data sets.

2 Background

An **RDF stream** S is a potentially unbounded sequence of timestamped informative units (d_i, t_i) ordered by the temporal dimension, where t_i is the timestamp (as in [14, 19, 51], we

[5] https://dev.twitter.com/rest/public/rate-limiting

consider the time as discrete) and d_i is a set of RDF statements. An RDF statement is a triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$, where I, B, and L identify the sets of IRIs, blank nodes, and literals, respectively. An **RDF term** is an element of the set $T = I \cup B \cup L$.

RSP Query Languages [14, 19, 30, 51] extend SPARQL[6] with operators to cope with streams. They enable the registration of queries over RDF streams. RSP queries are evaluated in a continuous fashion, i.e., results are computed at different time instances as the data flows in the streams. Given a query q, the answer $Ans(q)$ is a stream, to which the results of the evaluations are appended. This work focuses on the RSP query languages that support the **time-based sliding window** operator $\mathbb{W}$, which is defined through the parameter ω, the width, and β, the slide, and generates a sequences of fixed windows, i.e., portions of S in a time interval $(o, c]$ [14, 19, 51]. Given a time-based sliding window and two generated consecutive windows W_i and W_{i+1}, defined in $(o_i, c_i]$ and $(o_{i+1}, c_{i+1}]$, two constraints hold: $c_i - o_i = c_{i+1} - o_{i+1} = \omega$ and $o_{i+1} - o_i = \beta$.

Let V be a set of variables (disjoint with I, B and L), graph patterns are expressions defined recursively as: 1) a basic graph pattern, i.e., a set of triple patterns $(ts, tp, to) \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$, is a graph pattern; 2) let P_1 and P_2 be graph patterns, $P_1\ JOIN\ P_2$ or $P_1\ UNION\ P_2$ is a graph pattern; 3) let P be a graph patterns and $u \in I \cup V$, $SERVICE\ u\ P$ or $WINDOW\ u\ P$ is also a graph pattern. Other graph pattern expressions are possible (e.g. OPTIONAL, FILTER) but are not presented for the sake of space.

Like SPARQL, the evaluation semantics of RSP Query Languages rely on the notion of **solution mapping**, i.e., a partial function that maps variables to RDF terms, i.e., $\mu : V \to T$. A full formalization of RSP Query Languages is in [30]. We briefly describe the semantics of `WINDOW`, `SERVICE`, and `JOIN` in RSP Query Languages. Evaluating a **`WINDOW`** clause results in the content of a sliding window, similarly to what GRAPH does in SPARQL, which refers to the content of a named graph in the data set. The **`SERVICE`** retrieves mappings from SPARQL endpoints by submitting a graph pattern [4]. **`JOIN`** can be formally defined as: let $dom(\mu) \subset V$ be the set of variables mapped by μ, two mappings μ_1 and μ_2 are **compatible** (denoted with $\mu_1 \sim \mu_2$) if they assign the same values to the common variables, i.e., $\forall v \in dom(\mu_1) \cap dom(\mu_2), \mu_1(v) = \mu_2(v)$. We name **joining variables** the elements in $dom(\mu_1) \cap dom(\mu_2)$.

As explained, this paper focuses on queries containing the graph pattern:

$$(WINDOW\ u^W\ P^W)\ JOIN\ (SERVICE\ u^S\ P^S),$$

where P^W and P^S are two graph patterns that share one or more variables, u^S is the address of a service BGD in remote and u^W is an IRI denoting a sliding window operator $\mathbb{W}$ defined through ω, β and applied to a stream S.

Local view. Existing RDF stream engines leverage a nested loop join strategy to fetch data from BGD. It follows that evaluating the above graph pattern can be expensive: each request to BGD has a latency, computational and, possibly, financial cost. In the SPARQL endpoint of our experiments (see Section 6), each invocation takes $4.6ms$. Hence, during one second, it can only accommodate up to 200 requests. In real scenarios, SPARQL

[6] C.f. https://www.w3.org/TR/sparql11-query/ for additional reference.

endpoints are exposed over Internet, and each quest can take more than $500ms$ [60]. For this reason, we previously proposed to use a local view $\mathcal{R}$ to store the result of P^S in the RSP engine [26]. $\mathcal{R}$ stores the results of the `SERVICE` clause so that the engine computes the results of the query without invoking the SPARQL endpoint of BGD at each evaluation. However, given that the content of BGD changes over time, the mappings in $\mathcal{R}$ become outdated, and the evaluation of the `SERVICE` clause produces different solution mappings leading to wrong results. Therefore, each mapping $\mu^{\mathcal{R}} \in \mathcal{R}$ can be classified as *fresh* or *stale*: $\mu^{\mathcal{R}}$ is *fresh* at time t, if it is contained in the result set by evaluating the `SERVICE` clause over BGD at t; it is *stale* otherwise (i.e., if BGD changes, it produces different results when evaluating of the `SERVICE` clause over $\mu^{\mathcal{R}}$ and the remote BGD). In the following, we assume that mappings in BGD change with fixed intervals. This happens, e.g., in data warehouses, where updates are scheduled, or in data generated by sensors or automatic processes, where data is updated with fixed interval. As in [27], we define the freshness of an answer $Ans(q)$ as $\frac{|fresh(Ans(q))|}{|Ans(q)|}$.

Maintenance process. To ensure the freshness of the local view over time, we introduce a maintenance process MP that refreshes a portion of $\mathcal{R}$. MP selects a set of mappings $\mathcal{E} \subseteq \mathcal{R}$ to be refreshed within each evaluation of the queries over BGD. The design of MP is the key to the freshness of $Ans(q)$: if the process correctly identifies the stale mappings and puts them in $\mathcal{E}$, then both the freshness of $\mathcal{R}$ and $Ans(q)$ increase. Note, however, that if the number of refresh queries sent to BGD is too high, the presence of $\mathcal{R}$ does not bring any advantage. In practice, MP has to consider (i) Quality of Service requirements associated with the query, e.g., responsiveness; (ii) system reactiveness, e.g., each evaluation should terminate before the next one starts; (iii) constraints imposed by the BGD providers on the number of requests during a time interval. We capture these aspects by introducing a notion of **refresh budget** value Γ, defined as the number of refresh queries that can be sent to BGD in a given time period without violating the above constraints. We assume that Γ covers n evaluations, and denote with $\gamma = \frac{\Gamma}{n}$ the maximum refresh budget available in one evaluation.

3 Related Work

Traditional databases usually materialize remote BGD locally. Sophisticated optimizations of retrieving remote data on-demand have been introduced to improve availability, scalability and query processing performance [33, 39, 49]. The drawback of materialization is that local data becomes stale when the remote data changes. Those works are neither in stream processing context nor considering budget constraints on remote access.

In Complex Event Processing (CEP), the incoming events not only need to be matched with specified event patterns, but also need to be enriched [42, 78]. During enrichment, it usually needs to access remote BGD through APIs defined by service providers [43]. These API providers usually apply constraints on the number of accesses to restrict the massive loads of requests, as the computation and communication costs involved are shown to be intensive. Given the repetitive nature of the access to BGD [53], caching techniques can improve on response latency. However, when a cache becomes outdated, refreshing it

raises the trade-off between latency and freshness [1]. More remote accesses could provide fresher response, but take longer time. Authors in [49] address this trade-off in a web setting, where updates of the remote BGD are pushed into the system [39]. However, this work does not consider the constraints of service providers or the view maintenance without updates being pushed into the system.

In RDF processing, SPARQL 1.1 standardizes the access to remote BGD by introducing the federated extension [4] and the `SERVICE` clause. Broadly, there are two ways of accessing BGD: either one pulls the whole data into the query processor [50] or one 'federates' query-execution and transfers the data for individual operations over the network [44], defining new join strategies that can efficiently process both local and remote data [50]. Extending static RDF processing, RSP technologies deal with data of different velocity and variety. C-SPARQL [14] performs query matching on subsets of the information flow defined by windows. CQELS [51] implements its native query operators, which can be adaptively optimized to improve performance. MorphStream [19] allows querying relational data streams over a set of stream-to-ontology mappings. All those systems are optimized for processing streams. They support the `SERVICE` clause as described above but do not consider budget-constrained updates in the local view. Hence, our solution is orthogonal to these and other RSP engines.

Our previous work [26] studied the maintenance process of local view for querieswhere each mapping in the `WINDOW` clause joins with exactly one mapping in the `SERVICE` one. In this paper, we tackle a more general join relationship between `WINDOW` and `SERVICE` clauses, i.e., we extend the 1:1 join relationship to M:N and propose a flexible budget allocation method that further improves the maintenance process.

4 Problem Definition

Given the graph pattern expression P^S in the `SERVICE` clause, we define two sets of variables: first, $V^{SR} \subset var(P^S)$ contains the variables in $var(P^S)$ that are related to the changing part in BGD. In other words, V^{SR} captures the dynamicity of BGD and contains the information needed to construct the refresh queries that are sent to remote BGD. Second, V^{SN} are the common variables that join the P^S and P^W clauses, i.e., $V^{SN} = var(P^S) \setminus V^{SR}$. We model the relationship between V^{SR} and V^{SN} as a bipartite graph. The maintenance process MP exploits the graph to identify the candidate set $\mathcal{E}$ for refreshing. The MP builds a bipartite graph (*maintenance graph*, Figure 1) out of $\mathcal{C}$, which is a subset of $\mathcal{R}$. Mappings in $\mathcal{C}$ are (1) stale and (2) belong to the candidate set of the current window (i.e., they have compatible mappings in the result set Ω^W of the `WINDOW` clause). The maintenance graph has signature $G^{\mathcal{C}} = (\Omega^{SN}, \Omega^{SR}, E)$, where Ω^{SN} (Ω^{SR}) is the set of mappings with domain V^{SN} (V^{SR}), and E are the mappings $\mu^{\mathcal{R}}$ in $\mathcal{C}$, modeled as edges connecting elements of Ω^{SN} and Ω^{SR}.

Different subqueries in P^S have different optimization goals. In this work, we consider: 1) P^S is a Basic Graph Pattern (BGP) query; 2) P^S is an aggregate query[7].

[7] We assume that the aggregation is performed locally in the query processor and not in the remote BGD. It happens, e.g., when BGD is not SPARQL 1.1 compliant.

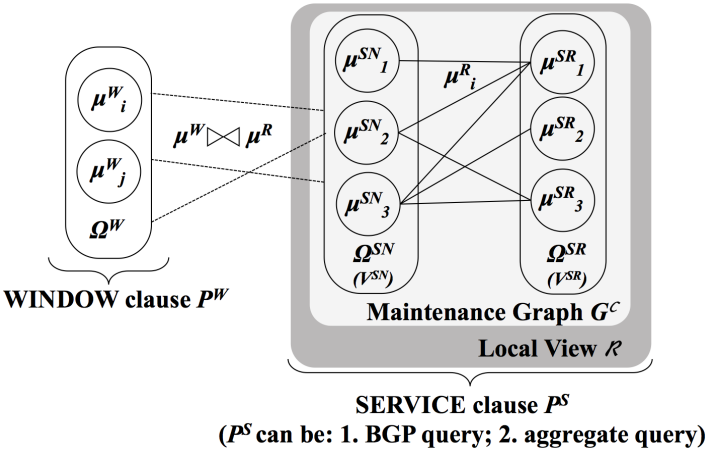


Fig. 1: `WINDOW`/`SERVICE` clauses and the Maintenance graph.

Case 1: P^S is a BGP query. By differentiating V^{SR} and V^{SN}, we split μ^R into two mappings $\mu = \mu^{SR} \cup \mu^{SN}$ such that $dom(\mu^{SR}) \subseteq V^{SR}$ and $dom(\mu^{SN}) \subseteq V^{SN}$. As P^S is a BGP query, each mapping μ_k^R consists a μ_i^{SN} and a μ_j^{SR}. Updating one μ_j^{SR} can ensure all its corresponding μ_k^R are fresh. As an example, consider the graph in Figure 1, where $\mathcal{C} = \{\mu_1^{\mathcal{R}}, \ldots, \mu_6^{\mathcal{R}}\}$. Ω^{SR} contains the mappings with the variables in V^{SR}, i.e., $\{\mu_1^{SR}, \mu_2^{SR}, \mu_3^{SR}\}$ (on the right); Ω^{SN} contains the other mappings, i.e., $\{\mu_1^{SN}, \mu_2^{SN}, \mu_3^{SN}\}$ (in the middle). The mappings in $\mathcal{R}$ are encoded as the edges in E (e.g., (μ_1^{SN}, μ_1^{SR}) represents $\mu_1^{\mathcal{R}}$). Updating μ_1^{SR} will make all its three corresponding mappings to be fresh: (μ_1^{SN}, μ_1^{SR}), (μ_2^{SN}, μ_1^{SR}), and (μ_3^{SN}, μ_1^{SR}). Given Ω^W (on the left) as the solution of the `WINDOW` clause and γ as the refresh budget at the current iteration, the maintenance process can be summarized as: refreshing which subset of Ω^{SR} can maximize the number of fresh join results between μ^W and μ^R? Formally, it can be modeled as the following optimization problem:

$$\text{Sub. } u_j^{SR} = 0 \text{ or } 1 \quad \forall j = [1, |\Omega^{SR}|] \tag{1}$$

$$\sum_{j=1}^{|\Omega^{SR}|} u_j^{SR} \leq \gamma \tag{2}$$

$$f_i^{SN} = \sum \mu_j^{SR} \quad \forall \mu_j^{SR} : (\mu_i^{SN}, \mu_j^{SR}) \in E \quad \forall i = [1, |\Omega^{SN}|] \tag{3}$$

$$c_i^{SN} = |\{\mu^W : \mu^W \in \Omega_W \wedge \mu^W \text{comp. with} (\mu_i^{SN}, \mu_j^{SR})\}| \ \forall i = [1, |\Omega^{SN}|] \tag{4}$$

$$\text{Max. } \sum_{i=1}^{|\Omega^{SN}|} f_i^{SN} * c_i^{SN} \tag{5}$$

The optimization is subject to: in Formula (1), the value of u_j^{SR} shows whether the j-th stale mapping is updated ($u_j^{SR} = 1$) or not ($u_j^{SR} = 0$). The total number of updates is limited by γ, as in Formula (2). Formula (3) defines f_i^{SN} as the number of fresh mappings μ_i^{SN} will have. Each μ_i^{SN} may have several related μ_j^{SR}. By summing all its refreshed μ_j^{SR}, we can have the total number of fresh mappings for μ_i^{SN}. As discussed above, this is because each updated μ_j^{SR} will produce one fresh $\mu_k^{\mathcal{R}}$ (μ_i^{SN}, μ_j^{SR}) . Overall, Formula (1) to (3) give the total number of fresh μ^R in the `SERVICE` clause. Since each $\mu^{\mathcal{R}}$ may have several compatible mappings in the `WINDOW` clause, Formula (4) introduce c_i^{SN} to represent the

number of compatible mappings of $\mu_k^{\mathcal{R}}$ in the window. Finally, our optimization goal is to maximize the total number of join results between `WINDOW` and `SERVICE` clauses, which could be defined as the product of c_i^{SN} and f_i^{SN}, as shown in Formula (5).

Case 2. P^S **is an aggregate query.** In this case, the maintenance graph $G^{\mathcal{C}}$ is constructed as the previous case: Ω^{SN} contains mappings with variables used for join, and Ω^{SR} contains mappings with dynamic values. However, Ω^{SR} in this case does not directly participate in the join, but are needed for aggregation.

Consider the example in Figure 1: $\mathcal{C} = \{\mu_1^{\mathcal{R}}, \mu_2^{\mathcal{R}}, \mu_3^{\mathcal{R}}\}$: $\mu_1^{\mathcal{R}}$ contains the value of the aggregate variables by using the data stored in μ_1^{SR}; $\mu_2^{\mathcal{R}}$ has an aggregate computed from μ_1^{SR} and μ_3^{SR}; $\mu_3^{\mathcal{R}}$ is computed from μ_1^{SR}, μ_2^{SR} and μ_3^{SR}. The edges, in this case, represent the mappings required to compute the aggregates, e.g., (μ_2^{SN}, μ_1^{SR}) and (μ_2^{SN}, μ_3^{SR}) indicate that the mapping $\mu_2^{\mathcal{R}}$ should be computed by using both the fresh values of μ_1^{SR} and μ_3^{SR}. The maintenance problem is still to choose a subset of μ^{SR} to maximize the fresh join results. However, in this case, updating one μ_j^{SR} cannot ensure its corresponding μ_k^R is fresh. To have a fresh μ_k^R, we need all its related μ^{SR} to be fresh. Therefore, the problem can be modeled as:

$$\text{Sub. } u_j^{SR} \leq 1 \quad \forall j = [1, |\Omega^{SR}|] \tag{6}$$

$$\textstyle\sum_{j=1}^{|\Omega^{SR}|} u_j^{SR} \leq \gamma \tag{7}$$

$$f_i^{SN} = \textstyle\prod \mu_j^{SR} \quad \forall \mu_j^{SR} : (\mu_i^{SN}, \mu_j^{SR}) \in E \quad \forall i = [1, |\Omega^{SN}|] \tag{8}$$

$$c_i^{SN} = |\{\mu^W \colon \mu^W \in \Omega_W \wedge \mu^W \text{comp. with}(\mu_i^{SN}, \mu_j^{SR})\}| \ \forall i = [1, |\Omega^{SN}|] \tag{9}$$

$$\text{Max. } \textstyle\sum_{i=1}^{|\Omega^{SN}|} f_i^{SN} * c_i^{SN} \tag{10}$$

The constraints in Formula (6) and (7) are same with Case 1. Formula (8) uses f_i^{SN} to model the fact that the i-th mapping μ_i^{SN} is fresh ($f_i^{SN} = 1$) iff all its related μ^{SR} are refreshed. For example, to have a fresh result of μ_2^{SN}, both μ_1^{SR} and μ_3^{SR} have to be 1; otherwise, $f_i^{SN} = 0$. Formula (9) is same with Case 1. Finally, the objective function in Formula (10) maximizes the number of fresh mappings produced by the join.

Overall, both Case 1 and 2 can be treated as binary integer programming problems. However, Case 2 can be seen as an extension of the knapsack problem, which is NP-hard, e.g., packing a μ^{SN} has a cost (the number of its μ^{SR}). We can only afford a certain number of μ^{SR}, but need to maximize the number of μ^{SN}. Furthermore, after choosing a μ^{SN} and its related μ^{SR} to pack, those μ^{SR} might contribute to other μ^{SN}. Therefore, choosing different μ^{SR} will have different influence on the following decisions. Currently, there is no optimal way to find the best subset of μ^{SR}.

5 Maintenance Algorithms

In this section, we propose a set of budget allocation algorithms. Section 5.1 proposes two greedy algorithms, SBM_{BGP} and SBM_{Agg}, for the problems in Case 1 and 2, respectively. They aim at maximizing the freshness of the current slide evaluation. Because the sliding

window operator supplies information about future evaluations (i.e., elements stay in the window for different periods), Section 5.2 shows how to exploit this information to improve the maintenance process. Section 5.3 discusses how to flexibly manage the budget to optimize the overall response freshness. The basic idea is to uniformly allocate Γ to n evaluations (i.e., $\gamma = \lfloor \Gamma/n \rfloor$). When it is worth well, the solution trades the current remote accesses for the future ones.

5.1 Selectivity-Based Maintenance (SBM)

To maximize the number of fresh join results, we propose the SBM$_{BGP}$ algorithm for Case 1, where P^S is a BGP query; and the SBM$_{Agg}$ for Case 2, where P^S is an aggregate query. In both cases, we start from the maintenance graph $G^{\mathcal{C}}$ defined above.

SBM$_{BGP}$. The objective function of Case 1 (Formula (5)) aims at maximizing the number of fresh mappings produced by the join. Based on $G^{\mathcal{C}}$, SBM$_{BGP}$ first computes a score for each mapping $\mu^{SR} \in \Omega^{SR}$, which represents the total number of the fresh join mappings that would be generated if μ^{SR} is updated:

$$score_{SBM}(\mu^{SR}) = \sum_{\mu_i^{SN}:(\mu_i^{SN},\mu^{SR})\in E} c_i \tag{11}$$

Based on the selectivity of μ^{SR}, the number of results it will have equals to the sum of each its connected μ^{SN} times μ^{SN}'s compatible mappings in the window. Then, SBM$_{BGP}$ picks μ^{SR} with the highest scores under the budget γ to refresh.

SBM$_{Agg}$. This case aims to maximize the number of fresh aggregate results. A mapping μ^{SN} produces a fresh aggregate result only if all its connected μ^{SR} are fresh. As discussed, fining the optimal set of μ^{SN} is a NP-hard problem. We propose a heuristic algorithm: SBM$_{Agg}$. It tries to utilize the budget on those "cheap" μ^{SN}, which connects to less stale μ^{SR}. Specifically, SBM$_{Agg}$ picks the mapping $\mu^{\bar{S}N}$ with the smallest amount of connected μ^{SR} and puts those μ^{SR} in $\mathcal{E}$. Then, $\mu^{\bar{S}N}$ and the mappings in $\mathcal{E}$ are removed from the maintenance graph $G^{\mathcal{C}}$, and a new iteration starts again. It ends when γ elements have been moved into $\mathcal{E}$. If the budget left $\gamma\prime$ is less than $\mu^{\bar{S}N}$, we will randomly choose $\gamma\prime$ amount of stale μ^{SR}.

5.2 The Impact-Based Maintenance (IBM)

The two SBM algorithms maximize the freshness of the current evaluation but do not consider future evaluations. As shown in [26], a maintenance process MP can take into account the sliding window and the changing frequency of the background data to have a prediction on what will be stale in future. We combine this idea with SBM to improve the performance of MP.

Before presenting the solution, we first introduce the concept of ranking data by a score based on two properties from [26], which quantify the impact of a mapping in future window evaluations. Consider a set of solution mappings Ω^W resulted from the evaluation of a `WINDOW` clause and a local view $\mathcal{R}$, where each mapping in Ω^W can have *only one* compatible mapping in $\mathcal{R}$.

The first property is the *remaining lifetime*, denoted with L. Let $\mu^{\mathcal{R}}$ be a mapping in $\mathcal{R}$, and let μ^{W} be its only compatible mapping in Ω^{W} computed at time t in a sliding window $\mathbb{W} = (S, \omega, \beta)$. The L value of μ^{S} at time t^{now} is computed as $\lceil (t+\omega-t^{now})/\beta \rceil$. It represents the number of evaluations, in which μ^{S} will be involved. For example, given a sliding window $\mathbb{W} = (S, \omega = 150, \beta = 30)$ and a mapping μ^{W} with timestamp $t = 100$, the L value of the compatible mapping $\mu^{\mathcal{R}}$ at time 100 is $L(\mu^{\mathcal{R}}, 100) = \lceil (100+150-100)/30 \rceil = 5$; at time 160, it is $L(\mu^{\mathcal{R}}, 100) = \lceil (100 + 150 - 160)/30 \rceil = 3$. The second property is the *number of evaluations before the next expiration*, denoted with B. Given a stale mapping $\mu^{\mathcal{R}}$, B represents the number of evaluations that $\mu^{\mathcal{R}}$ would be fresh, if refreshed now. B is computed as $B(\mu^{\mathcal{R}}, t^{now}) = \lceil (t^{exp} - t^{now})/\beta \rceil$, where t^{exp} is the next time on which $\mu^{\mathcal{R}}$ would become stale. t^{exp} is processed by exploiting the change rate interval information of $\mu^{\mathcal{R}}$. At time $t^{now} = 100$, the value of B is $B(\mu^{\mathcal{R}}, 100) = 3$, i.e., if $\mu^{\mathcal{R}}$ is refreshed now, it would remain fresh for the next three evaluations (evaluations at 100, 130, and 160; at 190, $\mu^{\mathcal{R}}$ will be stale).

Now, L and B can be combined to assign a *score* to the elements in $\mathcal{C}$ (i.e., the stale mappings in the local view currently involved). Intuitively, the *score* of the mapping $\mu^{\mathcal{R}}$ represents *how many future correct results are attainable if* $\mu^{\mathcal{R}}$ *is refreshed now.* The *score* of $\mu^{\mathcal{R}}$ at time t^{now} is computed as $score(\mu^{\mathcal{R}}, t^{now}) = \min\{L(\mu^{\mathcal{R}}, t^{now}), B(\mu^{\mathcal{R}}, t^{now})\}$. If $B(\mu^{\mathcal{R}}, t^{now}) < L(\mu^{\mathcal{R}}, t^{now})$ $\mu^{\mathcal{R}}$, it can generate at most $B(\mu^{\mathcal{R}}, t^{now})$ fresh join mappings, before it becomes stale while remaining in the window; otherwise, it generates $L(\mu^{\mathcal{R}}, t)$ fresh join results and will leave the window before it becomes stale. Based on this *score*, we extend the two SBM algorithms so that they also consider the future impact of a refresh. Given the maintenance graph $G^{\mathcal{C}} = (\Omega^{SN}, \Omega^{SR}, E)$ as defined in Section 4 (M:N bipartite graph), the extensions, namely IBM_{BGP} and IBM_{Agg}, can cope with the stale mappings μ^{SR} appearing in different mappings $\mu^{\mathcal{R}}$ of the local view.

IBM_{BGP}. We assign a score for the stale mappings in Ω^{SR}, as with SBM_{BGP}. The formula proposed above for B is still valid for the elements in Ω^{SR}. However, L cannot be directly associated with mappings in Ω^{SR} because they are related to the mappings computed by the `WINDOW` clause Ω^{W} through Ω^{SN}.

$$L(\mu^{\mathcal{R}}, \mu^{W}, t^{now}) = \lceil (t_{\mu^{W}} + \omega - t^{now})/\beta \rceil \tag{12}$$

$$score(\mu^{\mathcal{R}}, \mu^{W}, t^{now}) = \min\{L(\mu^{\mathcal{R}}, \mu^{W}, t^{now}), B(\mu^{SR}, t^{now})\} \tag{13}$$

$$score(\mu^{\mathcal{R}}, t^{now}) = \textstyle\sum_{\mu^{W}} c.w.\mu^{\mathcal{R}} score(\mu^{\mathcal{R}}, \mu^{W}, t^{now}) \tag{14}$$

$$score_{IBM_{bgp}}(\mu^{SR}, t^{now}) = \textstyle\sum_{\mu^{\mathcal{R}}=(\mu^{SN}, \mu^{SR}) \in E} score(\mu^{\mathcal{R}}, t^{now}) \tag{15}$$

IBM_{BGP} associates the remaining lifetime L to the pair of compatible mappings $(\mu^{\mathcal{R}}, \mu^{W})$ as defined in Formula (12): the function takes into account the arriving time $t_{\mu^{W}}$ of μ^{W} as well, in order to cope with the fact that there are multiple compatible mappings for a μ^{SR}. This extension allows defining a score for each pair $(\mu^{\mathcal{R}}, \mu^{W})$, as in Formula (13). It represents the number of fresh mappings that are potentially generated by joining $\mu^{\mathcal{R}}$ and μ^{W} in the current and the following evaluations, if a μ^{SR} is refreshed. Formula (14) computes the score of a mapping $\mu^{\mathcal{R}}$ in the local view, which sums the

scores of $\mu^{\mathcal{R}}$ with compatible mappings in Ω^W. Finally, IBM$_{BGP}$ assigns the score to the mappings in Ω^{SR} by Formula (15): it represents the total number of fresh join mappings that will be generated, if μ^{SR} is refreshed. We select γ mappings of μ^{SR} with the highest scores to refresh. Section 5.4 discusses why IBM$_{BGP}$ is a local optimal solution.

IBM$_{Agg}$. As discussed in Section 4, budget allocation, in this case, is a NP-hard problem. When future evaluations of the current data are considered, the complexity increases further, due to the additional level of combinatorial optimization. Therefore, IBM$_{Agg}$ exploits a *score* function to improve the basic SBM$_{Agg}$ algorithm. An aggregate value for a $\mu^{SN} \in \Omega^{SN}$ is fresh only when all the required mappings $\mu^{SR} \in \Omega^{SR}$ are fresh.

$$L(\mu^{SN}, t^{now}) = \lceil (\max_{t:\mu^W \in \Omega^W \wedge \mu^W \text{ c.w. } \mu^{SN}} \{t\} + \omega - t^{now})/\beta \rceil \tag{16}$$

$$score_{IBM_{agg}}(\mu^{SN}, t^{now}) = \min \{L(\mu^{SN}, t^{now}), \min_{\mu^{SR}:(\mu^{SN},\mu^{SR}) \in E} \{B(\mu^{SR}, t^{now}\}\} \tag{17}$$

IBM$_{Agg}$ computes the score of the mappings in Ω^{SN} when two or more of them have the same lowest amount of connected μ^{SR}. Specifically, Formula (16) computes the remaining lifetime of μ^{SN}, which takes the most recent timestamp of the compatible mappings of a μ^{SN} in Ω^W. Formula (17) reports the function to compute the score, which considers two factors: (1) μ^{SN} will continue to generate fresh mappings as long as all its related mappings μ^{SR} are fresh; (2) their compatible mappings of μ^{SN} still remain in the window.

5.3 Flexible Budget Allocation (FBA)

Above solutions only consider the fixed amount of refresh budget γ assigned in the current evaluation. However, fixing γ may be inefficient as the number of refresh requests changes over time. Saving current budget for future updates may improve result freshness, if the future ones can generate more results.[8] The semantics of the sliding window allow inferring how long each element in the current window will be involved in future joins. We propose FBA to allocate the refresh budget by considering both current and future evaluations. Specifically, FBA iterates from the current to the future ω/β slides (window length/slide length). At each iteration, it identifies the maintenance graph $G_i^{\mathcal{C}}$ and the stale data Ω_i^{SR}. It calculates the number of future fresh results for each μ^{SR} in every Ω_i^{SR} at their corresponding evaluation time and orders μ^{SR} by their scores. FBA allocates total $n \times \gamma$ budgets to the Top-$(n \times \gamma)$ μ^{SR} with the largest scores. Note that this set contains both current and future stale μ^{SR}. If the number of μ^{SR} in the current evaluation is less than γ, it means FBA delays the budgets of current μ^{SR} to some future ones.

5.4 Discussion

SBM$_{BGP}$ and IBM$_{BGP}$ are optimal for Case 1. For a BGP query, choosing the top-γ data in Ω^{SR} based on deg^{SR}, which is the number of μ^{SR}'s associated elements in Ω^{SN}. It

[8] We acknowledge that not all types of budget can be saved for future (e.g., a fixed amount of bandwidth cannot be saved). Other types of budgets, such as a supplier charges per request, a limited data plan, or limited power can be saved.

gives the local optimal solution at the current time without considering the future impact of Ω^{SR}. This is because the top-γ of Ω^{SR} is the set with the largest sum of deg^{SR}, since the sum of deg^{SR} exactly equals the number of fresh results. Therefore, SBM_{BGP} gives the local optimal solution. The same reason applies to IBM_{BGP}, where γ mappings with the largest *score* also gives the most results, as *score* accurately reflects the number of future results. Note that since the future elements in stream are not predictable (with certainty), there is no global optimal solution for BGP query.

Complexity. Both the SBM and IBM only consider data in the current evaluation. SBM_{BGP}/IBM_{BGP} visits each μ^W and μ^R to count the number of mappings and calculate scores for μ^{SR}, which both take linear time of $\mathcal{O}(|\Omega^W|+|\Omega^R|+|\Omega^{SR}|)$. Then, choosing the Top-γ mapping take $\mathcal{O}(|\Omega^{SR}|\log|\Omega^{SR}|)$ time. SBM_{Agg} takes $\mathcal{O}(|\Omega^{SN}|^2\log|\Omega^{SN}|)$ time, as whenever updating a μ^{SN}, we have to update all its related μ^{SR}. IBM_{Agg}, as an extension of SBM_{Agg}, has the same complexity. FBA has the same time complexity as IBM, since they have the same way of ranking and choosing data to refresh, except that IBM chooses data only in the current slide; FBA does this for a fixed number of future slides.

6 Experiments

Experiment environment. We implemented the maintenance process in a real RSP system: C-SPARQL [14]. The system registers continuous federated queries with `WINDOW` and `SERVICE` clauses (as in Section 2) and continuously evaluates the query per window on the incoming stream. Each evaluation joins the content of the current `WINDOW` with the results of the `SERVICE` clause. For evaluating the `SERVICE` clause, we have implemented a local view in C-SPARQL to cache remote BGD data (as in Section 4). Before executing the `SERVICE` clause, different maintenance algorithms will select a candidate set $\mathcal{E}$ from Ω^{SR} to refresh. For each data in $\mathcal{E}$, the `SERVICE` clause will request its fresh value from the remote server. We used Fuseki 2.0.0 as the remote BGD server and ran it with the C-SRAPQL engine on the same machine. The delay of each remote access under this setting is much smaller than querying an actual remote server.

Experiment data sets. We employ a *real* data set and several *synthetic* data sets to investigate the performance of our solutions. The real data set was recorded from Twitter. The synthetic ones were constructed by resembling the real one, but using a generator that can alter its characteristics. Each data set broadly contains three kinds of data: the remote BGD, the local view $\mathcal{R}$, and the input stream. We discuss the corresponding parameters and their values below.

The remote BGD. The main parameter of the remote data is its change interval ChR. In $real_{twitter}$ data that is collected in [26], the number of followers of 100 selected users is captured every minute for four hours. We noticed that the distribution of ChR is highly skewed: only few users have a very dynamic changing number of followers over time, while others are stable. Roughly, it resembles a Beta distribution with α=50 and β=1. In *synthetic* data, our data generator outputs data with different ChR-distributions. The generator has two parameters: the skewness of the distributions and the correlation with the selectivity of data elements in the local view. The latter allows us to control whether

data that changes more frequently can have either a higher or a lower selectivity. We will report the values of parameters in each experiment below.

The local view $\mathcal{R}$. In $\mathcal{R}$, we model the relationship between Ω^{SN} and Ω^{SR} as a bipartite graph. Therefore, we are mainly interested in the deg^{SR} of Ω^{SR}. After analyzing the $real_{twitter}$ data set, we observed a skewed distribution of the deg^{SR} that can be modeled as a Zipf distribution with a skewness parameter of 0.2. In the *synthetic* data, we can tune two aspects of deg^{SR}: the skewness and the correlation with the stream/remote data.

The stream and the sliding window. The *real* data set uses the stream of tweets containing the mentions of the monitored Twitter users described above. The *synthetic* data set generates the streaming data through a Poisson process [72]. To verify Hypothesis 3, the stream is generated with a non-homogeneous Poisson process, where the data arrival rate changes over time, e.g., $\lambda_i = 0.95\lambda_0 \cdot (i \bmod 2) + \lambda_0 \cdot ((i+1) \bmod 2)$, where λ_0 is the initially expected arrival interval and i is incremented along the time. The input query has a sliding window length of $\omega = 4$ seconds and slides every $\beta = 1$ second. Each experiment has 50 evaluations, and the first 10% is used as a warm-up period.

We first use synthetic data sets to verify our hypotheses and study the performance of our algorithms. The performance and the computational overhead on the real data set are reported as well. The average response freshness is used as the Key Performance Indicator (KPI). As discussed in Section 2, it is the ratio of fresh results to the total number of results, within each evaluation. The number of fresh results is acquired by comparing the current result set to the corresponding set acquired by the original C-SPARQL engine[9], where all results are fresh, since it queries BGD without budget constraints.

The baseline algorithms. We choose two baseline algorithms: 1) Least Recently Update (LRU), which selects the least recently updated *stale* data from $\mathcal{R}$; 2) Random (RAND), which randomly chooses *stale* data from $\mathcal{R}$. All algorithms pick at most γ (the refresh budget) candidates to refresh.

Resulting synthetic data sets. The default settings of the synthetic data set are: Ω^{SN} and Ω^{SR} in $\mathcal{R}$ contains 50 data elements each. There are 1000 edges μ^R between Ω^{SN} and Ω^{SR}. Each μ^R randomly connects a pair of μ^{SN} and μ^{SR}. Every μ^{SR} has a change interval ChR randomly chosen from [100, 3000] ms. A stream trace generated from a Poisson distribution decides the arrival time of each μ^{SN}. For the Poisson distribution, each μ^{SN} chooses its λ (the expected arrival interval) randomly from [1000, 2000] ms. The default budget γ is 10.

6.1 Verifying hypotheses H1 and H2.

H1 and H2 are tested together by comparing the response freshness among RAND, LRU, SBM and IBM in both subquery cases:

Case 1. In the four settings of Figure 2, both SBM_{BGP} and IBM_{BGP} greatly improve the response freshness of the baselines by up to 93%. These different settings show how the performance improvement generalizes.

[9] http://streamreasoning.org/larkc/csparql/CSPARQL-ReadyToGoPack-0.9.zip

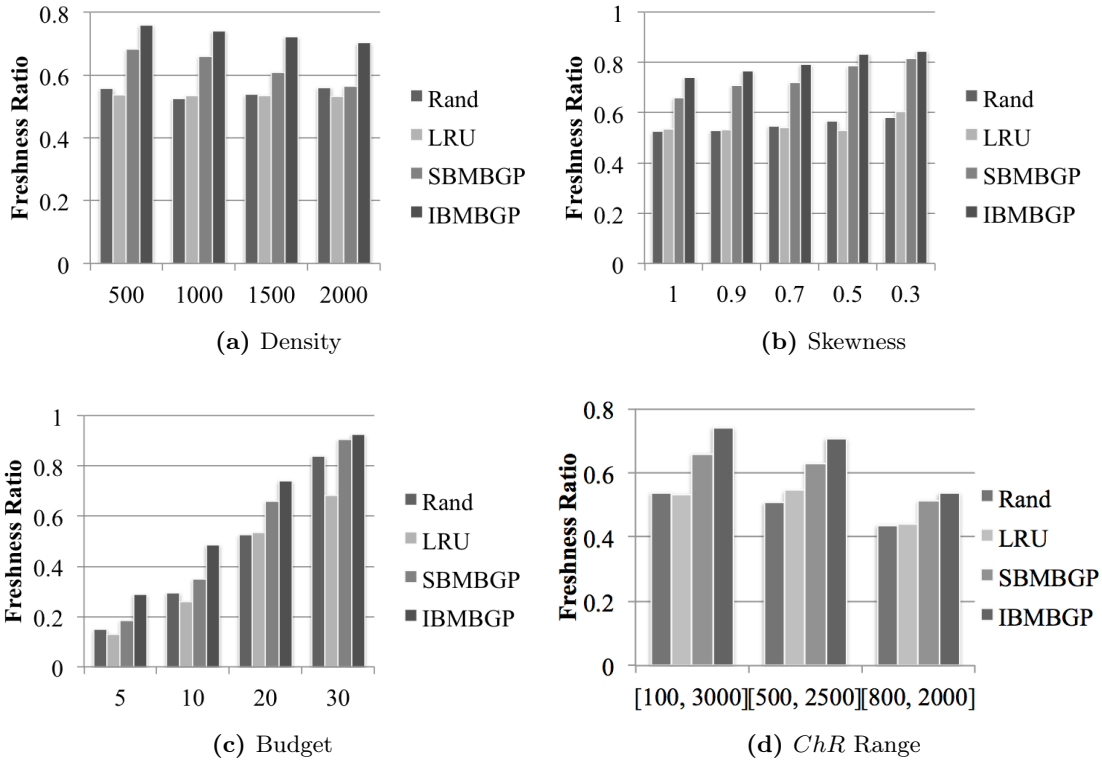


Fig. 2: SBM$_{BGP}$ and IBM$_{BGP}$ outperform baselines under different settings.

In Figure 2(a), we show the performance of using $\mathcal{R}$ with different densities, i.e., the number of edges $|\mu^R|$ in $\mathcal{R}$ is set to be 500, 1000, 1500, and 2000. Note that 2500 ($|\Omega^{SN}|\times|\Omega^{SR}|$) edges will form a fully connected $\mathcal{R}$. First, we observe that the performance of RAND and LRU remain roughly stable over different densities. The reason is that they select the refresh candidates $\mathcal{E}$ "blindly" without considering deg^{SR}—the selectivity of μ^{SR}. Therefore, the percentage of edges being updated remains the same for different densities. On the other hand, in higher densities, the performance improvement of SBM$_{BGP}$ and IBM$_{BGP}$ decreases a bit. The reason is that in a denser graph the difference of deg^{SR} among μ^{SR} becomes less significant. SBM$_{BGP}$ and IBM$_{BGP}$ always choose the μ^{SR} with the highest deg^{SR}, however, the percentage of the chosen μ^{SR} to the total number of μ^{SR} in $G^{\mathcal{C}}$ becomes smaller. Hence, SBM$_{BGP}$ and IBM$_{BGP}$ favor sparse graphs.

Figure 2(b) plots the performance on graphs with different distributions of μ^{SR}'s selectivity. We set the selectivity to follow different Zipf's distributions, with skewness parameter s to be 1 (uniform), 0.8 (slightly skewed), 0.5 (skewed), and 0.3 (highly skewed). Figure 2(b) shows that the performance improvement of SBM$_{BGP}$ and IBM$_{BGP}$ is more significant in skewed graphs. The reason is same with the second observation above: SBM$_{BGP}$ and IBM$_{BGP}$ refresh the μ^{SR} with the highest deg^{SR}. In a skewed graph, the percentage of the selected μ^{SR} increases, which leads to more fresh results. Therefore, SBM$_{BGP}$ and IBM$_{BGP}$ favor skewed μ^{SR} selectivity distribution.

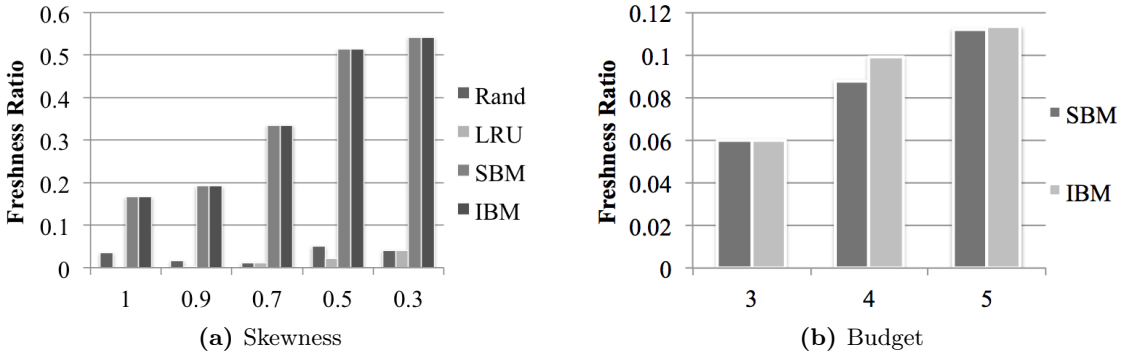


Fig. 3: SBM$_{Agg}$ and IBM$_{Agg}$ outperform baselines in subquery Case 2.

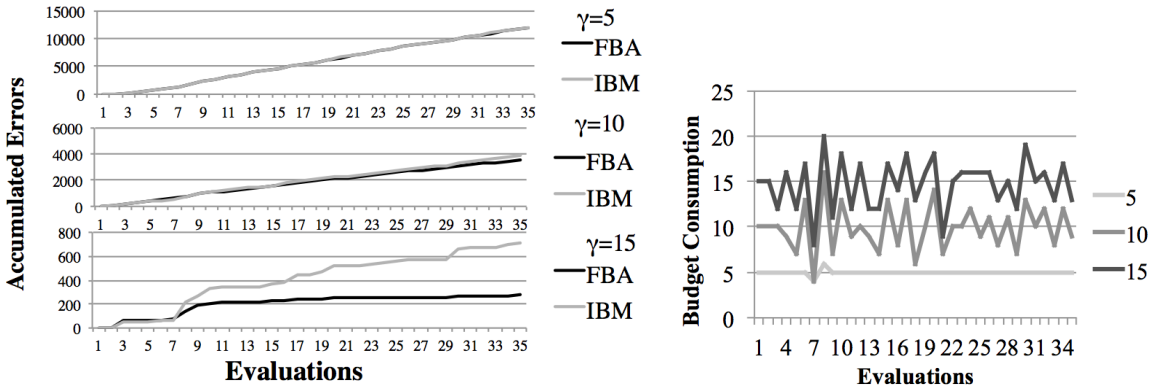
Figure 2(c) shows the performance with different budgets i.e., $\gamma = 5$, 10, 20, and 30. With a larger budget, the performance improvement of SBM$_{BGP}$ and IBM$_{BGP}$ becomes less. In an extreme case of having a large enough budget to cover most of the *stale* μ^{SR}, different subsets of $\mathcal{E}$ do not affect the freshness anymore. Therefore, SBM$_{BGP}$ and IBM$_{BGP}$ can achieve significant improvement with less budget. The above three experiments verify **H1**: considering the selectivity deg^{SR} of μ^{SR} enables choosing better candidates for refreshing and improves response freshness.

Regarding H2, Figure 2(d) shows the performance results of BGD change intervals that are randomly chosen from different ranges: [100, 3000], [500, 2000], and [800, 1200] ms. We can make these comparisons: first, IBM$_{BGP}$ always has a higher freshness than SBM$_{BGP}$. Second, having a wider range for ChR leads to better improvement in IBM$_{BGP}$. The reason is that IBM$_{BGP}$ chooses μ^{SR} with larger "impact", i.e., larger score, since the score indicates that μ^{SR} makes more results in the current and future slides. Therefore, this experiment verifies **H2** and shows that IBM$_{BGP}$ favors larger ranges of change intervals.

Case 2. When P^S is an aggregate query, in all of the above cases, we observed similar performance improvements of SBM$_{Agg}$ over RAND and LRU. To save space, we just show the results with different skewnesses to demonstrate the performance in Figure 3(a). Besides the freshness improvement, we notice that in most cases IBM$_{Agg}$ performs similarly with SBM$_{Agg}$. This is because IBM$_{Agg}$ is designed to be at least as good as SBM$_{Agg}$. Only when several μ^{SN} have the same amount of connected μ^{SR}, SBM$_{Agg}$ will choose the one with the lowest score. Furthermore, for different μ^{SN}, when the overlapping between their associated μ^{SR} is small, the chance of μ^{SN}s have different scores is larger and the effect of SBM$_{Agg}$ is, therefore, more significant. Figure 3(b) investigates this by plotting the results of a special case: a very sparse graph (100 edges) and a tiny budget, e.g., 3 to 5. In these cases, IBM$_{Agg}$ outperforms SBM$_{Agg}$ by up to 12.5%.

6.2 Verifying hypothesis H3

We compare the performance of FBA with IBM in Case 1 with three refresh budgets, $\gamma = 5, 10, 15$ in Figure 4(a). They track the accumulated number of stale results over time. By increasing the budget, the gap between FBA and IBM becomes more significant.



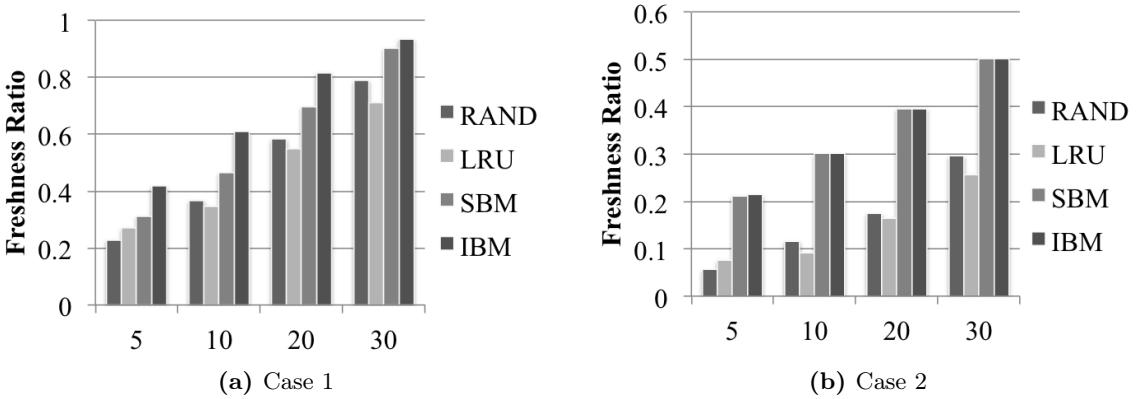
(a) Accumulated error **(b)** Budget usage of FBA over time

Fig. 4: The performance of FBA under different budgets.

Furthermore, when $\gamma = 15$, after the first 15 iterations, FBA makes the accumulated stale result increases very slowly, i.e., the freshness ratio of the answer is almost 100%, while IBM still keep producing stale results. To explain the improvement, Figure 4(b) plots the actual amounts of budget that are consumed over time. For IBM, the consumption of budget will always be a vertical line for different budgets. For FBA, when $\gamma = 5$, the line fluctuates a bit. With larger budgets, the lines fluctuate more. It shows that FBA moves budgets between different slides to improve freshness.

Results on a real data set. Figure 5 plots the results on a real dataset with different budgets for both cases. We can observe that IBM always achieves the best freshness and SBM also outperforms the two baseline algorithms. When we decrease the budget, the performance improvement of IBM and SBM increases. These results confirm our findings in Figure 2(c).

Computational overhead. We finally report the computational overhead and the average remote access delay. Under the default setting $\gamma = 20$, the total latency of a slide



(a) Case 1 **(b)** Case 2

Fig. 5: SBM and IBM outperform baselines in a real dataset.

evaluation is about 94.4 ms. The delay of querying the BGD server accounts for 92 ms on average (4.6 ms per request); the computational overhead is only about 2.3 ms (2.5% of the overall latency). Note that, the current setting has the remote BGD server running locally. When requests are sent over the Internet, the computational overhead will become even more negligible while the performance gain will become more substantial.

7 Conclusions and Future Work

In this paper we studied the problem of accessing remote background data (BGD) from an RDF Stream Processing (RSP) context. When BGD is large, stored remotely, and/or changing over time, accessing it can be expensive, waste resources, and deteriorate the response time. Hence, a local view is often used to speed up the BGD accesses, but maintaining it is often subject to refresh budget constraints. This paper proposes to efficiently allocate the budget for refreshing the local view. Specifically, our solution relies on a bipartite graph to model the join between stream data and BGD. It exploits the graph structure to improve response freshness for two kinds of `SERVICE` subqueries: a BGP query (Case 1) and an aggregate query (Case 2). Our solution, SBM, exploits a set of basic algorithms that leverage the selectivity of the join between the stream and the background data. Experiments show that it can significantly improves the response freshness up to 25% compared to baseline algorithms (i.e., RAND and LRU). An also introduce an improved approach, IBM, that takes the future impact of refreshes into account and improves the performance up to 55.6% over the SBM. Finally, we propose the FBA optimization that flexibly allocates budget considering not only the current but also future data. As a result, FBA significantly improves over all other solutions and maintains a freshness of close to 100% even in the light of limited update budget.

Our findings have the following limitations: first, our algorithm for Case 2 relies on a greedy heuristic approach. We hope to investigate a more advanced approximate approach in the future. Second, the current approach focuses on BGP. Some SPARQL operators (e.g., OPTIONAL) can introduce new challenges and require non-trivial extensions of our model. Third, current FBA only moves the budget forward for future evaluations. An alternative is to do the opposite: trade future budget for current refresh demand, and adapt the two-way budget allocation according to stream.

Even in the light of these limitations we believe that this paper highlights an important problem in RSP—the joint evaluation of stream and BGD under budget constraints—and provides solutions for different subqueries. As such it paves the way for truly scalable RSP systems in real-world environments, where the integration of stream and BGD is ubiquitous.

Acknowledgements

This research has been partially funded by Science Foundation Ireland (SFI) grant No. SFI/12/RC/2289, EU FP7 CityPulse Project grant No.603095 and the IBM Ph.D. Fellowship Award 2014 granted to Dell'Aglio.

Distributed Stream Consistency Checking

This chapter is based on a submission to the:

2017 IEEE International Conference on Big Data(Big Data 2017)

Distributed Stream Consistency Checking

Shen Gao[1], Daniele Dell'Aglio[1], Jeff Z. Pan[2], and Abraham Bernstein[1]

[1] Department of Informatics, University of Zurich, Switzerland
[2] Department of Computer Science, The University of Aberdeen, United Kingdom

Abstract. Streamed data is improving people's daily life. Various smart city-related streams are combined to suggest suitable travel plans according to real-time traffic situations, accidents and weather conditions. However, the noise inherent to these streams may cause wrong results. While noise has usually been largely studied in settings where streams have a simple schema (e.g., time series), few solutions have been proposed to cope with streams characterized by complex data structures.
This paper studies how to check consistency over large amounts of complex streams in a distributed fashion. The methods we propose exploit reasoning to assess if portions of the streams are compliant to a reference conceptual model. To achieve scalability, our methods are designed to be deployed in state-of-the-art distributed stream processing platforms such as Apache Storm or Heron. Our first baseline method consists of calculating the closure of Negative Inclusions (NIs) for an ontology and registering the NIs as stream queries. The second method compiles the ontology into a stream processing pipeline to evenly distribute the workload. By analyzing the trade-offs between the two methods, we further propose a cost model that can be used for parameter tuning. Experiments show that our solution improves the system throughput of the baseline method by up to 139%.

1 Introduction

Big data velocity is changing our lives. City-related sensor streams are improving our drive experience, with real-time information about navigation, traffic, and events. Similarly, data continuously generated on the Web enables the usage of search engines as up-to-date newspapers, where breaking news and articles are listed at the top of query results.

However, noise can lead to wrong and unexpected results. Errors in traffic sensors may lead to wrong representations of the current status of a city, causing–in the worst case–to new jams. It is worth noting that noise is inevitable in real streaming settings, so technological solutions to identify and cope with it are necessary. A possible way to handle noise is setting data integrity constraints, such as data range filters that exclude outliers generated by sensors with hardware failures. However, other types of noise may be very hard to be detected, in particular when constraints are defined w.r.t. a complex conceptual model that describes the underlying data. The problem we study in this article is how to assess the *consistency* of a set of streams w.r.t. a fixed and known a-priori conceptual model.

Being in a stream processing scenario, an important requirement is *responsiveness*: the consistency assessment process must be responsive and quick when analyzing newly arrived data. Batch-based solutions are generally not efficient, because they usually involve high latency and process data overlapped between batches multiple times. Our solutions take into account Stream Processing Engines (SPEs) techniques [22], proposed in the literature to cope with the velocity dimension.

Checking if a (static) dataset is consistent w.r.t. a conceptual model is a well-known problem in data and knowledge management [5, 54]. Moving to the streaming setting,

consistency checks in the presence of data streams has only partially considered so far [82]. However, such studies do not tackle the volume dimension of the problem and assume that the input stream can be managed as a whole. Moreover, they do not consider the recentness dimension, typical of stream processing use cases: the more recent the data, the more relevant it is. The first challenge we have to cope with is *how to model the problem of stream consistency over streams?* We consider a description logic, DL-lite$_{core}$, to express the conceptual model and to describe the consistency constraints. DL-lite$_{core}$ is one of the description logics with the lowest complexity level, but it is still expressive enough to express subclass relations, as well as class disjunctions.

The last challenge is related to the *volume* of the data: the size of the stream may be too large to fit in one computing node. A way to overcome this limitation is to perform the consistency check in a distributed fashion. Therefore, we ask ourselves *how to distribute the consistency reasoning task over streams?* There are many state-of-the-art Distributed Stream Processing Engines (DSPEs), e.g. Apache Flink, Apache Storm and Twitter Heron. When using these engines, users usually need to compile the business logics into a processing workflow. We study how it is possible to build a consistency checking procedure on top of DSPEs.

Summarizing, the main contributions of this article are:

- we formally define the consistency checking task over streams, based on DL-lite$_{core}$;
- we propose two methods to assess the consistency of the streams w.r.t. a fixed conceptual model. These methods are designed to work in a distributed environment; and
- we conduct a comparative study, based on the LUBM benchmark using Twitter Heron, to investigate the performance of the proposed solutions

The rest of the paper is organized as follows: Section 2 reviews related work. Section 3 introduces the background of semantic streams, consistency check, and distributed stream processing. Our solutions and their optimization are presented in Section 5. Section 6 provides evaluation results of our solution. Section 6 closes with final remarks.

2 Related Work

This section first discusses the related work on consistency checking for a knowledge base. Then, it introduces distributed stream processing and RDF stream processing.

2.1 Consistency Checking of a Knowledge Base

Consistency checking is one of the most important tasks in reasoning. Many tools and plug-ins for reasoners have been proposed. [11] translates an ontology to the language that can be executed by a logic programming engine. [58] translates an ontology for checking consistency on instances of a knowledge base. There are also various reasoning systems designed for specific ontologies. For example, the SnoRocket System is designed for the SnoMed Terminology [59]. Authors in [76] propose a reasoning system for the YAGO ontology. We aim to propose a generic reasoning approach over streams.

In static settings, when the underlying logic is FOL-rewritable, a common way to check consistency is through query rewriting. There are various query rewriting techniques that have been proposed for the DL-lite$_{core}$ family [16, 47, 66]. Our work re-uses some results of these studies and proposes new solutions for a streaming setting. Recently, [64, 65] propose the adoption of machine learning techniques to achieve fact consistency checking on a large amount of instances in a knowledge base. These methods first apply standard reasoning techniques to label inconsistent data instance. Then, they learn a model from the labeled data and use this model to classify new instances. While these approaches are very interesting, they involve the cost of labeling the dataset, and they do not guarantee a 100% accuracy.

Currently, most of the research on reasoning assumes a static KB, there are a few works on incremental reasoning and evolving ontologies [52, 69]. However, these works did not discuss reasoning of a distributed stream setting.

2.2 Distributed Stream Processing

Stream processing relies on the idea of managing data *in motion*, by performing tasks in a continuous fashion. In the recent years this paradigm has gained popularity due to the rise of Distributed Stream Processing Engines (DSPEs) that perform stream processing in clusters and cloud services. One of the first DSPEs to gain popularity has been Storm [79]. Storm relies on the notion of *topology*, a processing workflow where each element is named *task*. Storm automatically handles the distribution of tasks to computing nodes. Recently, Twitter proposed Heron [48] as a successor of Storm. Heron overcomes some limitations of Storm, such as limited performance monitoring, impossibility to deploy in clusters with heterogeneous nodes and complexity in debugging. Other DSPEs are Apache Flink [20], Apache Samza, and Google MillWheel [2]. Each of these systems has different design goals. For example, Apache Samza embeds a key-value store to manage state between processing node. Apache Flink emphasizes the combination of batched-based and stream-based processing paradigms. Google MillWheel deals with out-of-order data arrival in the stream. All of the above systems share the idea of letting the user define a workflow of operators, similarly to Apache Storm's topologies.

2.3 RDF Stream Processing

A recent effort to study the stream processing paradigm in the semantic web is RDF Stream Processing (RSP). The idea is to use (and extend) existing semantic web technologies to process sequences of timestamped RDF data. Solutions like C-SPARQL [14] and CQELS [51] propose the adoption of *sliding windows* to create time-varying views over the streaming content, to process through the typical relational algebra operators, such as the ones proposed by SPARQL. On the other hand, solutions like EP-SPARQL [3] and INSTANS [70] propose to verify time-relation constraints over the elements of the stream, usually in a close time interval. The existing RSP solutions have been developed in centralized systems. Compared to DSPEs, existing RSP engines show limitations in scalability, as well as in managing data characterized by impressive velocity and volume.

3 Preliminaries

This section first introduces DL-lite$_{core}$ as the Description Logic (DL) considered in this study. Based on a static DL-lite$_{core}$ ontology, we introduce the definitions about evolving ontology and its representation through RDF streams. Finally, we present the main concepts of distributed stream processing that we employ to build our solutions.

3.1 Static DL-lite$_{core}$ Ontology

Static knowledge bases can be represented using ontologies. In this paper, we focus on DL-lite$_{core}$, a description logic of the DL-Lite family [6]. The basic building blocks of DL-lite$_{core}$ are named concepts (denoted with A) and named roles (denoted with P). They can be used to build basic concepts C and basic roles R as follows:

$$C := \bot \mid A \mid \exists R \qquad R := P \mid P^{-}$$

A DL-lite$_{core}$ ontology is composed of a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$, denoted with $\langle \mathcal{T}, \mathcal{A} \rangle$. The TBox $\mathcal{T}$ contains axioms about concepts and roles, in the forms of:

$$C_1 \sqsubseteq C_2 \quad \text{or} \quad C_1 \sqsubseteq \neg C_2$$

The former axiom, also known as Positive Inclusion (PI), indicates that C_1 is a subclass of C_2 (e.g., *Student* is a subclass of *Person* can also be denoted as $Subclass(Student, Person)$). The latter one represents a Negative Inclusion (NI), which expresses that two classes are disjoint (e.g., there cannot be an instance of *Person* and *Organization*). A NI can be alternatively denoted as $Disjoint(C_1, C_2)$.

The ABox $\mathcal{A}$ contains assertions about individuals, which can be either $C_k(x_1)$ or $R_k(x_1, x_2)$. If a NI inferred by $\mathcal{T}$ is violated by assertions of the ABox $\mathcal{A}$ (e.g., $Disjoint(C_1, C_2)$ in $\mathcal{T}$ and $C_1(a)$ and $C_2(a)$ in $\mathcal{A}$), then the knowledge base is inconsistent. As discussed in [16], only the NI assertions can lead a DL-lite$_{core}$ knowledge base to inconsistency.

3.2 RDF and DL ontology streams

An *RDF stream* $S = ((d_1, t_1), \ldots, (d_n, t_n), \ldots)$ is a potentially unbounded sequence of timestamped informative units (d_i, t_i) ordered by the temporal dimension, where t_i is the timestamp (we consider the time as discrete) and d_i is an RDF statement. An RDF statement is a triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$, where I, B, and L identify the sets of IRIs, blank nodes and literals, respectively. An *RDF term* is an element of the set $T = I \cup B \cup L$. RDF streams are used to serialize DL ontology streams, defined as above.

We define *DL ontology streams* starting from the notion of *evolving ontology* as defined in [52], which captures the dynamics of a knowledge base. Given a discrete time interval $[m, n]$, a DL ontology stream $\mathcal{O}^n_m$ is a pair $\mathcal{O}^n_m = (\mathcal{T}, \mathcal{A}^n_m)$, where $\mathcal{T}$ is a TBox and $\mathcal{A}^n_m$ is a stream of ABoxes from time n to m. We refer with $\mathcal{A}^n_m(i)$ to the ABox (*snapshot*) at time i associated to $\mathcal{A}^n_m$. Similarly, we refer with $\mathcal{O}^n_m(i)$ to the pair $(\mathcal{T}, \mathcal{A}^n_m(i))$, which is a static ontology.

When handling a stream, it may be important to analyze a portion of data items together, (e.g., count the occurrences of a fact over a time interval). Therefore, inspired by the research on stream and event processing [22], we employ the notion of *window*, which is an operation that selects a portion of items in the stream. Let $\mathcal{O}^n_m$ be a DL ontology stream . The application of a window W over $\mathcal{O}^n_m$ results in:

$$\mathcal{O}^c_o = W(\mathcal{O}^n_m, \omega, c) = (\mathcal{O}^n_m(o), \ldots, \mathcal{O}^n_m(c)) = \\ = (\mathcal{T}, \langle \mathcal{A}^n_m(o), \ldots, \mathcal{A}^n_m(c) \rangle) = (\mathcal{T}, \mathcal{A}^c_o)$$

where ω is a natural number representing the *size* of the window, and c is the time on which the window is applied. The following constraints hold: $o = \max\{m, c - \omega\}$ and $c \leq n$.

Finally, we define the *window content* as the union of the axioms contained in the stream snapshots, i.e., $u(\mathcal{A}^n_m) = \bigcup^n_{i=m} \mathcal{A}^n_m(i)$. Given a DL ontology stream $\mathcal{O}^n_m$, it follows that $\langle \mathcal{T}, u(\mathcal{A}^n_m) \rangle$, is a knowledge base.

3.3 Distributed Stream Processing Engines (DSPEs)

Processing large amount of streams usually relies on Distributed Stream Processing Engines (DSPEs). In the following, we introduce basic concepts of DSPE, adopting the nomenclature of Apache Storm and Heron.

In Figure 1, a user compiles the processing logics into a *logical topology*, which is a Directed Acyclic Graph (DAG) composed of *spout* and *bolt* nodes. Spouts (S) provide input data to bolts; they emit data to downstream nodes and are typically used to connect a topology with external data sources such as Web services or data brokers. Bolts (B_i) embed the logics of processing the stream: they manipulate data from upstream nodes and emit results to downstream nodes.

Each edge of the topology represents a data stream. Data streams flow through the topology as sequences of *tuples*, i.e. sets of property-value pairs. While defining the logical topology, the user should declare the tuple format (i.e., the set of properties) for every edge.

In addition to the structure of the logical topology, the user should provide information to let the platform deploy the logical topology to a computing cluster. First, the user needs to define how many instances of each node should be deployed. These instances are named *task instances* or, shortly, tasks. Moreover, for each edge in the logical topology, the user should define a grouping strategy (e.g., a hashing function), which is used by the DSPE to partition the stream among the tasks. For this reason, given a stream a subset of the tuple attributes acts as a *key*. The DSPE uses the logical topology and the configuration parameters to decide how to distribute the tasks among the available computing servers. This results in a *physical topology*, as depicted on the right of Figure 1.

3.4 Problem definition

The problem we investigate is to assess if a stream is consistent in a time frame. That means, given a stream $\mathcal{O}^n_m$ and a time interval of ω time units, we aim to verify if

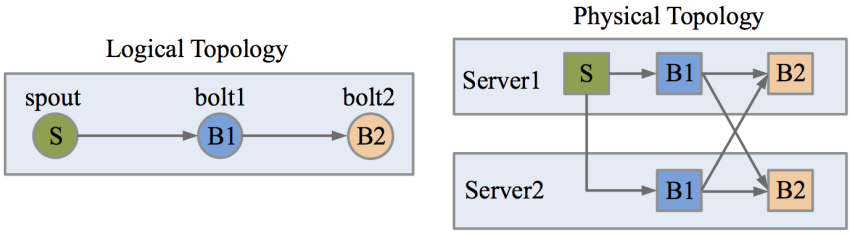


Fig. 1: Deploying a logical (left) to a physical (right) topology. The spout (S) and bolt (B1, B2) are instantiated to be a spout task and two bolt tasks on two computing servers.

$\langle \mathcal{T}, u(W(\mathcal{A}^n_m, c, \omega))\rangle$, is consistent for every $c \in [m, n]$ by using a DSPE. It is worth noting that when ω is large, i.e., $\omega \geq n - m$, the problem becomes the incremental consistency assessment over the whole content of the stream rather than the content captured in a sliding portion.

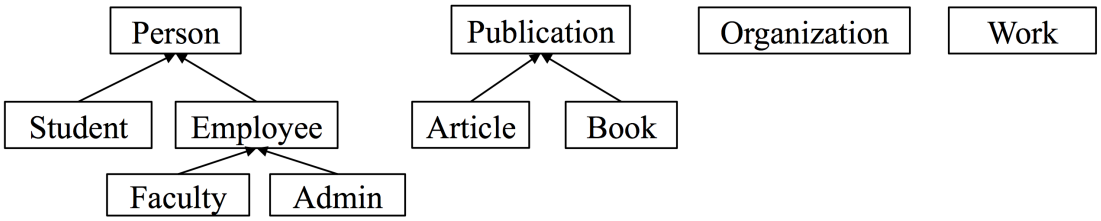
4 Solution

Given a DL ontology stream $\mathcal{O}^n_m = (\mathcal{T}, \mathcal{A}^n_m)$, our solution aims to compile the TBox $\mathcal{T}$ of a DL ontology as consistency checking operations that can be executed as a topology of a DSPE. This section first discusses several important components of a topology in DSPE. Then, it introduces two methods to compile a TBox. The first is a baseline algorithm named Negative Inclusions Topology Method (NTM). The second is an improvement of the first algorithm, named Pipeline Topology Method (LTM). To guide the parameter tuning in LTM, we also propose a cost model to address the trade-offs in our solutions. This section ends with a discussion of the current limitations.

Example 1. Figure 2 depicts the TBox $\mathcal{T}_{ex}$ of the running example we consider in this section. It is based on the LUBM benchmark, which is also used for the evaluation. Each node in the figure is a class, while edges denote positive inclusion axioms (PIs), e.g., the axiom $Subclass(Student, Person)$ is represented by the nodes $Student$, $Person$ and the edge between them. The lower part of Figure 2 contains the negative inclusion axioms (NIs) (e.g., Axiom (1) indicates that an instance of $Student$ cannot be an instance of $Publication$). In total, there are ten classes, six PIs, and ten NIs.

Tuple. With reference to the RDF stream model defined in Section 3.2, each instance in an input stream is a tuple with the format of a quadruple (s, p, o, t). The adoption of DL-lite$_{core}$ implies that a tuple can describe either a role or a class assertion. Class assertions $C(x)$ and $\exists R(x)$ are represented as (x, isA, C, t) and (x, isA, C_R, t), respectively, where t denotes the snapshot $\mathcal{A}^n_m(t)$.

Stream. A stream S_C contains all the tuples that state a class assertion of $C(a)$, where a is a generic individual (e.g., S_{Person} is a stream that has instances like $(Bob, isA, Person, t_1)$). A special stream, $S_{C_{inc}}$, indicates the stream of instances found to be inconsistent. For example, given the Axiom (1) in $\mathcal{T}_{ex}$, if there



Person⊑¬Publication (1) Person⊑¬Organization (2) Person⊑¬Work (3)
Publication⊑¬Organization (4) Publication⊑¬Work (5) Organization⊑¬Work (6)
Student⊑¬Employee (7) Article⊑¬Book (8) Faculty⊑¬Admin (9) Faculty⊑¬Student (10)

Fig. 2: Tbox $\mathcal{T}_{ex}$ of the example ontology. The upper part contains the positive inclusion axioms (PIs); the lower part contains the negative inclusion axioms (NIs).

are two instances $(Bob, isA, Personent, t_1)$ and $(Bob, isA, Publication, t_1)$, the tuple $(Bob, isFoundToBe, InConsistent, t_1)$ should be appended to $S_{C_{inc}}$. We manage the role assertions as follows. Given $R(a, b)$, two class assertions are generated: $\exists R(a)$ and $\exists R^{-}(b)$, where R^{-} indicates the inverse property of R. Consequently, two inputs are added to two streams.

Input and output. The input to a topology consists in the sequence of streams associated to class assertions. Given $\mathcal{T}_{ex}$, the input is the set of streams $\mathcal{S} = \{S_{C_1}, ..., S_{C_n}\}$, where each stream corresponds to one class in $\mathcal{T}_{ex}$ (e.g., $\mathcal{S} = \{S_{Student}, ..., S_{Work}\}$ and $|\mathcal{S}| = 10$). The topology outputs $S_{C_{inc}}$, which reports all inconsistent instances.

Spouts and bolts. For the sake of illustration, we assume that there is only one spout serving as stream broker, i.e., it collects streams from different sources and emits them as $\mathcal{S}$.

A topology consists of a set of bolts $\mathcal{B} = \{B_1, B_2, \ldots, B_n\}$. A bolt B_i encodes a set of operations $\{o_1, o_2, \ldots, o_n\}$. An operation o_i takes as input a set of streams $\mathcal{S}^{o_i}_{input} = \{S_{C_1}, ..., S_{C_n}\}$ and emits output streams $\mathcal{S}^{o_i}_{output}$. Intuitively, an operation encodes a part of the consistency check logics.

We further define two kinds of operations: inference and conjunction operations An *inference operation*, $o^{\rightarrow}$, is similar to a mapping function, which takes one stream S_{C_1} as input and emits one output stream S_{C_2}:

$$o^{\rightarrow} : S_{C_1}(x) \rightarrow S_{C_2}(x).$$

Its main usage is to encode subclass inferences.

Example 2. Given the $\mathcal{T}_{ex}$ in Example 1, an $o_i^{\rightarrow}$ derives a new instance of $Person$ for each instance in the $Student$ stream, since $Student$ is a subclass of $Person$:

$$o_i^{\rightarrow} : S_{Student}(x) \rightarrow S_{Person}(x).$$

The second kind of operations is *conjunction operation*, $o^{\cap}$. It is similar to a join function, which takes as input a set of streams and emits an output stream. It is in the form of:

$$o^{\cap} : S_{C_1}(x) \wedge S_{C_2}(x) \wedge \ldots \wedge S_{C_n}(x) \rightarrow S_{C_{output}}(x)$$

We mainly use it to check inconsistencies.

Example 3. Given the NI Axiom (1) of $\mathcal{T}_{ex}$ in Example 1, the operation $o_i^{\cap}$ is defined as:

$$o_i^{\cap} : S_{Person}(x) \wedge S_{Publication}(x) \rightarrow S_{C_{inc}}(x).$$

Differently from an inference operation, a conjunction operation requires the presence of a set of axioms to trigger the underlying rule. To guarantee the correct behaviour, its implementation requires caching instances of each input streams. The window definition discussed in Section 3.2 is employed for this purpose. A window caches instances when they arrive and deletes them when their associated time instant expires. In this way, a conjunction operation can join instances arrived at different moments within the window. Note that a conjunction operation is much more expensive than an inference operation in terms of both computational and memory cost.

Example 4. Regarding to Example 3, each stream of S_{Person} and $S_{Publication}$ is associated with a window. When an instance $S_{Person(Bob)}$ arrives, $o_i^{\cap}$ first checks whether $S_{Publication(Bob)}$ is cached. If yes, $o_i^{\cap}$ outputs $S_{C_{inc}(Bob)}$, since the join between these two instances violates the NI Axiom (1); otherwise, $Person(Bob)$ is cached for future use.

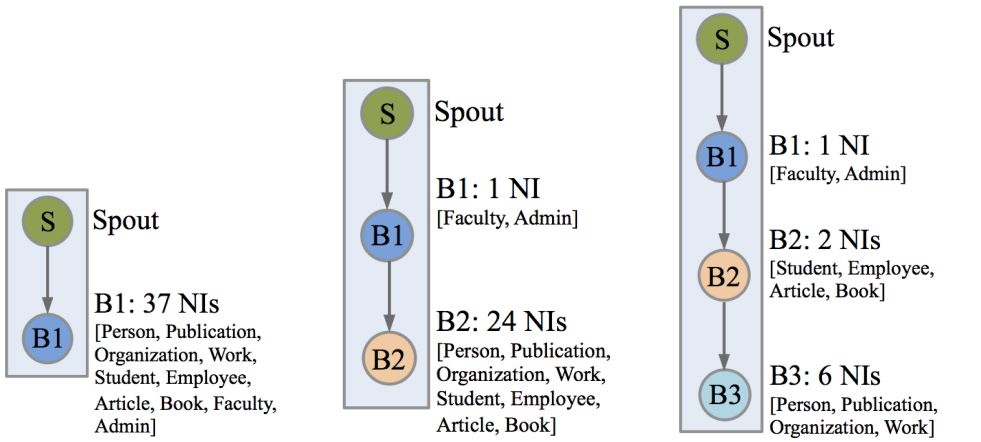
Lastly, if an instance does not belong to the input of neither inference nor conjunction operations, the bolt simply forwards it to the downstream bolts. This is needed to guarantee that all the inconsistencies are detected.

4.1 Overview of the solutions

Figure 3 depicts three alternative topologies that are compiled from $\mathcal{T}_{ex}$. The simplest method, in Figure 3a, first exhaustively finds all the possible NIs (all possible disjointness axioms). Then, it compiles each of them into a *conjunction operation* and assigns them to a bolt.

Example 5. The bolt B_1 contains all possible 37 distinct NIs that can be derived from the $\mathcal{T}_{ex}$ (Axioms $Disjoint(C_1, C_2)$ and $Disjoint(C_2, C_1)$ are considered to be same). There are nine NIs related to $Work$ that should be checked, since for each instance of $Work$, it needs to ensure the disjointness with all of the rest nine other classes.

This topology has the shortcoming that one single bolt contains all the NIs to evaluate, which makes it a performance bottleneck. Therefore, we define a second method to arrange the NIs in a hierarchical fashion and assign them to multiple bolts. The bolts in this new topology are chained as a pipeline, where each bolt is responsible for a limited amount of classes and their related NIs. This method introduces as parameter the number of bolts to be used. The topologies in Figure 3b and 3c show two different topologies with two and three bolts, generated according to this method, which produce the same result (with different performance).



(a) NIs Topology **(b)** Pipeline Topology 1 **(c)** Pipeline Topology 2

Fig. 3: The NIs Topology (NTM) and the Pipeline Topology (LTM). The workload in the last bolt of the three topologies reduces as the length of the pipeline increases.

Example 6. In Figure 3b, bolt B_1 is assigned to process only two streams *Faculty* and *Admin*, as well as the one NI between them. In addition, B_1 has two *inference operations* that convert the two streams into one *Employee* stream and send it to bolt B_2. B_1 also forwards all other streams to B_2. Bolt B_2 does not need to consider *Faculty* and *Admin* anymore. The disjointness between *Faculty* (*Admin*) and the rest classes is still ensured by the disjointness between *Employee* and other classes. Comparing to the one in Figure 3a, this topology has the cost of adding an extra bolt B_1, however, it greatly reduces the number of NIs to be handled in the last bolt from 37 to 24. Assuming that the same incoming stream rate for each class, the bottleneck of the last bolt is significantly reduced. Furthermore, the total number of NIs in both B_1 and B_2 is also smaller than the number of NIs in Figure 3a. The topology in Figure 3c uses three bolts. Bolt B_2 handles $Disjoint(Student, Employee)$ and $Disjoint(Article, Book)$, while bolt B_3 handles only four input streams (associated with *Person*, *Publication*, *Organization*, and *Work*) and the six NIs among them. This topology has more overhead, since streams are forwarded through B_1 and B_2.

By comparing the three topologies in Figure 3, we can observe that the total 37 NIs in B_1 of Figure 3a is reduced to be 24 and 6 NIs in B_2 and B_3 of Figure 3b and Figure 3c, respectively. The cost of checking NIs is distributed along the pipeline topology to remove the bottleneck. On the other hand, this benefit comes with the cost of adding extra bolts, which leads to higher computation and communication costs. Essentially, this second method, named LTM, looks for a balance between the evenness of NIs distribution on the pipeline and the length of the pipeline. In Section 4.2 and 4.3, we go into details with the two methods, while Section 4.4 proposes a cost model to address this trade-off.

4.2 The NIs Topology Method (NTM)

As written above, the idea of NTM is simple: it finds all possible NIs that can be derived from the TBox (denoted by $SNI_{closure}$) and it creates a conjunction operation for each NI and associates them to one bolt.

To compute $SNI_{closure}$, given an NI in the TBox, the algorithm computes a list of subclasses for each class in the NI. The permutation of elements in subclass lists gives all possible combinations between subclasses, and hence the NI closure. For example, the closure of $Disjoint(Publication, Work)$ is: $Disjoint(Article, Work)$ and $Disjoint(Book, Work)$. Note that the algorithm only keeps the distinct NIs in $SNI_{closure}$. Each NI in the closure set is in the form of $Disjoint(C_1, ..., C_n)$ (e.g., $Disjoint(Faculty, Work)$), which can be implemented as a conjunction operation. Therefore, NTM can calculate $SNI_{closure}$ before runtime and convert each NI into a conjunction operation. As a result, the *NIs topology* consists of a spout and one bolt. As shown in 3a, B_1 performs all the conjunction operations and signals if inconsistencies are detected. When deploying this topology, multiple task instances partition the stream by the subject field in a tuple, e.g., given a tuple (a, isA, C, t), a is the key.

As shown in the experiment, the single bolt of NTM creates a bottleneck on the throughput. Authors in [16] show that the size of $SNI_{closure}$ is exponential to the size of TBox at worst. Consequently, there is an exponential number of operations in the single bolt.

We considered two possible remedies to the bottleneck problem, but neither of them solves it completely. The first remedy is to increase the number of tasks for the bolt. It increases the parallelization of the bolt, but cannot reduce the complexity of the bolt. When the stream rate increases, the workload on each task increases as well, which proposes the same problem. As shown in the experiment, this method cannot solve the problem completely.

The second possible remedy is to have a different topology layout. Consider a topology where the spout connects to two bolts B_1 and B_2 (B_1 and B_2 are not chained as a pipeline). However, this topology does not offer any advantages when processing the pairwise PIs between $Person$, $Publication$, $Organization$, and $Work$ (Axioms (1-6)), since these four streams cannot be split into two bolts. The topology can use bolt B_1 to process streams $Person$ and $Publication$, and bolt B_2 to process the others. However, bolt B_1 still needs the stream of $Organization(Work)$ in order to check consistency between $Person$ and $Organization(Work)$. Therefore, the streams of $Organization$ and $Work$ have to be duplicated to B1, and vice versa, to ensure the correctness. Therefore, we propose the pipeline topology–LTM.

4.3 The Pipeline Topology Method (LTM)

LTM organizes the NIs to be processed in multiple bolts of a pipeline, as shown in Figure 3b and Figure 3c. It has the benefit that each bolt is responsible for a subset of the streams and their related NIs. The following bolts in the pipeline do not consider streams processed by one bolt.

Algorithm 1 CompilePipelineTopology($\mathcal{T}$)

input : $\mathcal{T}$, The TBox of a given Ontology
output: $\mathcal{B}$, A pipeline of bolts filled with operations

1 $SNI_{closure}$ = ComputeSetOfNIClosure()
2 SNI_{root}=ComputeSetOfNIRoots($SNI_{closure}$)
3 $SNIGroups$=[] **while** SNI_{root}*.size() is not reducing* **do**
4 | $SNIGroups$.add(GetAGroupOfSNI(SNI_{root}))
5 | SNI_{root}.remove($SNIGroups$.getLast())
6 **end**
7 $SNIGroups$.add(SNI_{root})
8 $SNIInBolts$ = AssignSNIGroupsToBolts($SNIGroups$)
9 $processedClasses$ = []; $\mathcal{B}$= [] **for** *(i = 0; i< SNIInBolts.size(); i++)* **do**
10 | $\mathcal{B}$.add(b_i = new Bolt())
11 | b_i.add(GetConjOps($SNIInBolts$[i]), $processedClasses$)
12 | b_i.add(GetInferOps($SNIInBolts$[i], $SNIInBolts$[i+1]))
13 | SC=Set(C_i of all $Disjoint(C_1, ..., C_n) \in SNIInBolts[i]$)
14 | $processedClasses$.add(SC)
15 **end**
16 **return** $\mathcal{B}$

Algorithm 1 gives the procedures of compiling a pipeline topology. The algorithm takes the TBox $\mathcal{T}$ of the DL ontology stream as an input. By calling a standard classification service on $\mathcal{T}$, it derives all possible subclass axioms (e.g., $Faculty$ has two super classes: $SubClass(Faculty, Employee)$ and $SubClass(Faculty, Person)$). The output of the algorithm is a chain (pipeline) of bolts. Each bolt is filled with the operations it needs to perform. The algorithm develops in five steps: Steps 1-3 find the *essential* NIs, order them and split them into NI groups. Steps 4 and 5 assign these groups to bolts and generate the corresponding operations. The details of each step the algorithm are explained in below.

Step 1. As with NTM, Line 1 of Algorithm 1 computes a set of the closure for all NIs, denoted as $SNI_{closure}$.

Step 2. Based on $SNI_{closure}$, Algorithm 1 computes SNI_{root}, which is a set of all the *essential* NIs (Line 2). The intuition behind essential is the following. Given two NIs (e.g. $Disjoint(Faculty, Student)$ in Axiom (10) and $Disjoint(Student, Employee)$ in Axiom(7)), if each class in one NI is either a sub or an equivalent class of the other (e.g., $Faculty$ in Axiom (10) is a subclass of $Employee$ in Axiom (7); $Student$ in the two NIs are the same.), the algorithm can process the NI that contains the superclasses (e.g., $Disjoint(Employee, Student)$) and make an inference from the subclass to its superclass (e.g., o_i: $Faculty \rightarrow Employee$). For the example in Figure 1, SNI_{root} contains all the disjoint axioms except Axiom (10).

The detailed procedure of finding SNI_{root} is given in Algorithm 2. Lines 17-25 compare each pair of different NIs in $SNI_{closure}$. Line 21 tests whether all classes ($\{C_1, ..., C_n\}$) in the first NI is a sub or an equivalent class of the classes in the second NI ($\{D_1, ..., D_n\}$).

Algorithm 2 ComputeSetOfNIRoots($SNI_{closure}$)

input : $SNI_{closure}$
output: SNI_{roots}
17 $SNI_{toremove} = \{\}$ **foreach** $Disjoint(C_1, ..., C_n) \in SNI_{closure}$ **do**
18 $sub = \{C_1, ..., C_n\}$
19 **foreach** $Disjoint(D_1, ..., D_n) \in SNI_{closure}$ **do**
20 $super = \{D_1, ..., D_n\}$
21 **if** $\forall C \in sub\, \exists D \in super : C = D \vee SubClass(C, D)$ **then**
22 $SNI_{toremove}$.add($Disjoint(C_1, ..., C_n)$)
23 **end**
24 **end**
25 **end**
26 **return** $SNI_{closure} \setminus SNI_{toremove}$

If it is the case, the first NI can be marked as removable. Line 26 removes those marked NIs from $SNI_{closure}$ and returns the rest.

Step 3. This step is encoded In Algorithm 1 Lines 3-6, where it iteratively finds groups of NIs that can be processed before others. Intuitively, an NI can be processed before others if all its classes have no subclasses. For example, classes of NI Axiom (9) have no subclasses. It should be check at the beginning of a pipeline topology (e.g., Place it in bolt B_1 of Figure 3c. If placed in bolt B_3, both its input streams will have to be forwarded via B_1 and B_2). Such NIs can be found by counting the "in-degree". Figure 4a gives an example. In the figure, if there is a sub-to-super relationship between classes of two NIs, we draw an edge from the NI with the subclass to the NI with the super. The NIs with no incoming edges are the ones with zero in-degree. However, not all NIs with zero in-degree should be processed first. For example, $Disjoint(Organization, Work)$ should not be processed in bolt B_1 of Figure 3c, since both streams $Organization$ and $Work$ will be used again later in the pipeline by other NIs ($Organization$ is used to check disjointness with $Person$). If it is placed in B_1, every tuple of $S_{Organization}$ and S_{Work} still needs to be forwarded to B_3, where the axiom $Disjoint(Organization, Person)$ are checked.

Algorithm 3 gives the details of finding a group of NIs. The algorithm is adapted from the topological sorting. Each NI in the result fulfills both: 1) has zero in-degree; and 2) its classes are used by the NIs outside the group. Line 27 finds all the NIs that have zero in-degree and puts them into $SNI_{indeg=0}$. The rest of NIs are put into $SNI_{indeg\neq0}$. From the NIs in $SNI_{indeg=0}$, Line 33 finds those share common classes with the NIs in $SNI_{indeg\neq0}$ and puts them into $SNI_{toExclude}$. Algorithm 3 returns $SNI_{indeg=0}$ minus $SNI_{toExclude}$ as a group of NIs that can be processed ahead of others.

Step 4. Given the result of Step 3, each NI group can individually form a bolt in the pipeline. This, however, may create too many bolts, and each new bolt incurs extra overhead. To avoid creating excessive amount of bolts, Line 8 in Algorithm 1 combines NI groups. Several adjacent NI groups can be combined into one bolt (Step 5 handles the combined groups of NIs). Figure 4 gives two ways of combining NI groups to bolts to

Algorithm 3 GetAGroupOfSNI(SNI_{root})

input : SNI_{roots}
output: SNI_{group}
27 $SNI_{indeg=0} = GetNIZeroIndegree(SNI_{roots})$
28 $SNI_{indeg\neq 0} = SNI_{roots} \setminus SNI_{indeg=0}$
29 $SNI_{toExclude} = \{\}$
30 **foreach** $Disjoint(C_1, ..., C_n) \in SNI_{indeg=0}$ **do**
31 **foreach** $Disjoint(D_1, ..., D_n) \in SNI_{indeg\neq 0}$ **do**
32 **if** $\exists C_i : C_i = D_j$ **then**
33 $SNI_{toExclude}$.add($Disjoint(C_1, ..., C_n)$);
34 **end**
35 **end**
36 **end**
37 **return** $SNI_{indeg=0} \setminus SNI_{toExclude}$

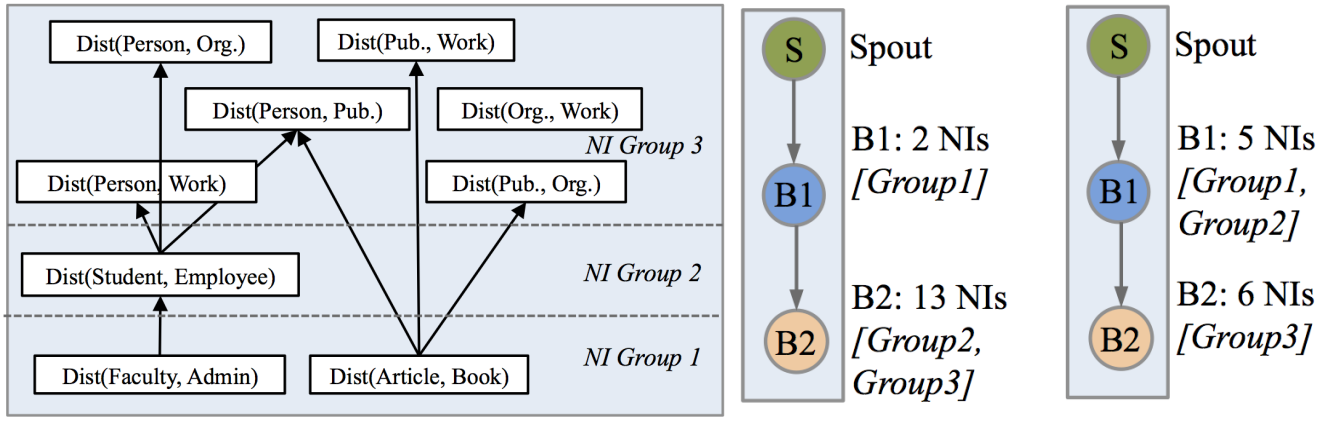
demonstrate the trade-offs. Figure 4a contains three NI groups separated by the dashed line. In Figure 4b, the left topology assigns Group 1 to bolt B_1, and Group 2 and 3 to bolt B_2. The right topology has Group 1 and 2 assigned bolt B_1, and Group 3 to bolt B_2. We observe that: although bolt B_1 on the right has more inference operations than B_1 on the left, the total number of NIs to be processed in B_2 is much smaller on the right than the left. When all input streams have the same stream rate, the workload distribution between B_1 and B_2 is more balanced in the right topology than in the left. Section 4.4 discusses what is the best way of assigning NI groups to bolts based on estimated cost.

Step 5. The last step (Line 9-16) in Algorithm 1 compiles the actual operations ($o^{\rightarrow}$ and $o^{\cap}$) for each bolt.

To compile *conjunction operations*, the algorithm computes the closure of all combined NI groups in order to ensure correctness. For example, in the right topology of Figure 4b, since these streams are processed together, the algorithm first computes their NI closure and then checks each of them (e.g., in the closure, the algorithm checks the disjointness of *Faculty* with *Admin*, *Article*, and *Book*; the disjointness of *Employee* with *Article*, and *Book*). In contrast to the topology in Figure 3c, where these two NI groups are not combined, neither bolt B_1 or bolt B_2 need to check the NIs involving *Faculty* with *Article* and *Book*, since *Faculty* is inferred as *Employee* and handled in B_2. Therefore, there is the trade-off on whether to combine NI groups: combining them may lead to many NIs in a bolt, but saves communication cost. The cost model in Section 4.4 addresses this trade-off.

The process of computing an NI closure in a bolt has been discussed in NTM. To avoid having duplicated NIs in different bolts, the closure needs to exclude the classes that have processed before (by passing the parameter *processedClasses*).

For compiling *inference operations*, the we need to find the subclass axioms between the classes of two consecutive bolts (B_i and B_{i+1}). For example, in the left topology in Figure 4b, bolt B_1 should includes the subclass axioms between B_1 and B_2 (e.g.,



(a) NI DAG and NI groups **(b)** Two ways of assigning NI groups into bolts

Fig. 4: Getting NI groups and assigning them to bolts.

$SubClass(Faculty, Employee)$ and $SubClass(Admin, Employee)$). Note that only the *direct* subclass axioms are considered. Axioms like $SubClass(Faculty, Person)$ are not included, since B_2 handles the NIs between $Employee$ and all other classes. Furthermore, bolt B_1 also needs to include all the subclasses of the classes in B_2 that are not participate in the NI roots. For example, if class $Employee$ (handled in B_2) has a subclass $FemaleEmployee$, which is not disjoint with either $Faculty$ or $Admin$, it still needs to be inferred at bolt B_1 to become $Employee$ so that the disjointness between $FemaleEmployee$ and other classes are checked in B_2. After getting the direct subclass axioms, each one of them can be compiled to an inference operation $o^{\rightarrow}$.

Result returned. Algorithm 1 returns an list of bolts that forms a pipeline. Each bolt is filled with the necessary operations to check the consistency of the input Tbox.

4.4 Cost Model

This section proposes a cost model that estimates the performance of both NTM and LTM topologies. The model supports the user decision process about the parameters to be used to generate the topology.

This study considers two main limiting factors of a topology: the CPU cost (computational cost) within a bolt and the network cost (communication cost) across bolts. Memory cost is also discussed in Section 4.5. The following discussion focuses on the bolts, since the spout only acts as a simple stream broker. The main Key Performance Indicator (KPI) is the system throughput (e.g., number of tuples that can go through a topology per time unit). We also report the results of processing latency in Section 5.

One of the most important factors that determines the throughput is the number of task parallelization. Consider NTM, the single bolt can be instantiated to n_{NTM} number of task instances. Let TP_{NTM} denote the throughput of the NTM topology and TP_{B_1} the throughput of one task instance. TP_{NTM} should be proportional to n_{NTM}:

$$TP_{NTM} = n_{NTM} \times TP_{B_1} \tag{1}$$

The throughput of a task instance TP_{B_1} can be modeled as:

$$TP_{B_1} = \frac{r_{S_{input}} \times t}{r_{S_{input}} \times t \times Cost(B_1)} = \frac{1}{Cost(B_1)}, \tag{2}$$

where $r_{S_{input}}$ is the aggregated stream rate for all input streams S_{input}. Note that we assume the stream rate is same for each class in the input. Section 4.5 discusses the case of input streams with different rates. During a time interval t, $r_{S_{input}} \times t$ gives the total number of tuples; $rate_{input} \times t \times Cost(B_1)$ gives the total amount of processing time that it needs. $Cost(B_1)$ is the average processing latency (CPU cost) for each tuple. By dividing the total number of tuples and the total processing time, we can drive the throughput TP_{B_1}. Intuitively, the throughput of a single task is inversely proportional to the processing latency $Cost(B_1)$. Each class in B_1 is involved in a different number of NIs. The cost for processing one class C_i is linearly related to number of conjunction operations it involves, $|\{o_{C_i}\}|$. Therefore, the average cost $Cost(B_1)$ is:

$$Cost(B_1) = \sum_{i=0}^{n} |\{o_{C_i}\}| / |C| \tag{3}$$

The above equations allow users to estimate the throughput of a NTM topology, TP_{NTM}.

Before extending the cost model for LTM, we first make the assumption that each bolt in LTM has the same number of deployed tasks (n_{LTM}). Setting different numbers of tasks for different bolts can potentially improve the performance; however, it creates a new problem of finding the best configuration. Authors in [32] propose a machine learning method to solve it. In this work, we do not consider this problem to highlight the trade-offs in consistency checking.

We extend the cost model from NTM to LTM in two ways. First, in LTM there are multiple bolts. The one with the minimal throughput is the critical one since, intuitively, the throughput of the pipeline can be as high as the bottleneck bolt. Therefore, we have:

$$TP_{LTM} = n_{LTM} \times min(TP_{B_1}, \ldots, TP_{B_n}) \tag{4}$$

Second, the throughput of an individual bolt in LTM is affected by the presence of inference operations. Their cost should be much smaller than the cost of conjunction operations, since it is a simple mapping operation. We introduce α as the ratio between the cost of inference to conjunction ($\alpha < 1$). On the other hand, if two bolts B_1 and B_2 in LTM have exactly the same workload, but B_1 is in front of B_2, B_2 should have a lower throughput than B_1. The reason is that each tuple has to go through B_1 and the network before reaching B_2. Therefore, to include the network cost, we add an extra penalty ρ ($\rho > 1$), which increases exponentially as depth of a bolt in the pipeline (e.g., B_k should have a penalty ρ^{k-1}, since a tuple has to be forwarded(processed) by the $n-1$ bolts in the front). By combining these two aspects, the cost of an bolt B_n in LTM can be modeled as:

$$Cost(B_k) = \rho^{k-1} \frac{\alpha \times \sum_{i=0}^{m} |\{o_{C_i}^{\rightarrow}\}| + \sum_{j=0}^{n} |\{o_{C_j}^{\cap}\}|}{|C|} \tag{5}$$

In practice, the values of α and ρ can be measured by running a benchmark topology.

As shown in the experiments, different ways of assigning NI groups to bolts lead to different performance. Given a Tbox, users can first compile a LTM topology with the highest estimated throughput. For example, the cost model can tell that the throughput of the right topology is higher than the left in Figure 4b. Then, users also compile a NTM topology, compare it with the LTM topology, and choose the one with higher throughput to deploy.

4.5 Limitations and Discussion

We acknowledge the following limitations of our solutions and briefly discuss possible solutions.

Extension to other DLs. The current work considers the DL-lite$_{core}$ logic. One of its closest extensions is DL-lite$_{horn}$ [6], which introduces the conjunction of concepts in the left operator, i.e. $C_1 \sqcap ... \sqcap C_n \sqsubseteq C$ ($Female \sqcap employee \sqsubseteq FemaleEmployee$). The conjunction operation we defined above natively supports this type of axioms. However, Algorithms 1, 2 and 3 need to be extended to cope with it.

Another extension is to consider $\mathcal{EL}^{++}$ [8, 9], since it is one of the most popular logics at the moment. Ontologies like SnoMed CT [71] and the OWL2 EL profile [61] are based on it. The most interesting aspect of this logic is the presence of transitive axioms, which may cause streams to form loops in a topology. A possible solution is to extract these axioms and manage them before or after the topology.

Memory cost. One shortcoming of LTM is its relatively high memory cost comparing with NTM. The reason is that an instance of a stream can be potentially cached multiple times at different bolts of a LTM topology. For example, in Figure 3b, stream $S_{Faculty}$ needs to be cached in B_1, since B_1 checks its NI with S_{Admin}. After stream $Faculty$ is inferred as $Employee$ and sent to B_2, its instances need to be cached again to check the NIs related to $Employee$. Our experiments also show this shortcoming. LTM should consider the memory limitation when assigning NI groups to bolts. This work assumes enough memory to emphases the CPU cost.

Changing stream rate. In reality, the stream rate for each input stream is different (e.g., $S_{Student}$ can have a much higher rate than $S_{Faculty}$). One way to refine our cost model is to assign a weight to each stream according to their input rate. Operations of streams with higher rates should have higher weights, since they consume more computing resource. Furthermore, when the stream rate changes over time, the pipeline topology should be adjusted accordingly in order to achieve the best performance.

Evolving TBox. The TBox is this work is static. However, a conceptual model may also evolve over time, which requires a revision of the deployed topology. At the best of our knowledge, no DSPE offers the feature of modifying topology at runtime. A typical—but naive—way to tackle this issue is to stop the topology, modify it (compile the new TBox), and restart. However, this solution usually leads to a downtime of the system. A promising direction under our investigation is to allow the bolts in LTM update the operations inside it. Operations can be encoded as parameters and feed to bolts on the fly. By changing

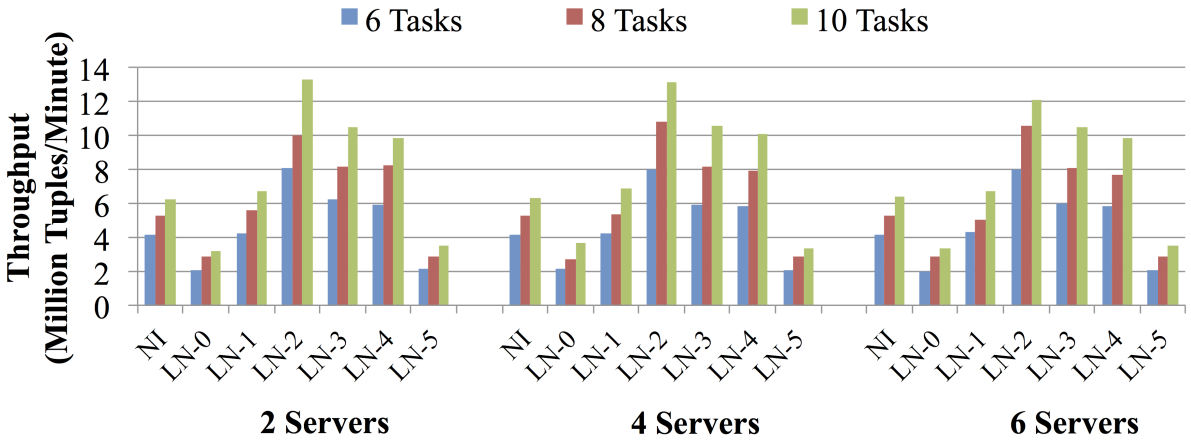


Fig. 5: LUBM results of a NTM topology and LTM topologies with two bolts in the pipeline.

these parameters during runtime, we will be able to execute a new version of the TBox without restarting the topology.

5 Experiments

This section experimentally compares NTM and LTM. We first introduce the experiment setup and the parameters and then report the results.

Experimental setup. We adapt the TBox of LUBM [40] to be compliant with DL-lite$_{core}$, which only keeps the positive and negative inclusions. A standard DL reasoner, HermiT [37], is employed to calculate the closure set of PIs.

Both LTM and NTM have only one task instance for the spout, which is implemented as a data generator that emits one input stream for each class in the TBox, as discussed in 4. In total, there are 43 streams in the input stream set SI. The generator emits each input stream at the same speed, fine tuned to target the system processing capability, as discussed in Sections 4.4 and 4.5. The throughput is determined by the *slowest* bolt in a LTM topology. We can compare the throughput of both topologies by measuring the number of tuples the system can process during a time interval. We use Heron 0.14.3 as the DSPE. All experiments run on a cluster of 20 machines, each machine having 128 GB RAM and two E5-2680 v2 at 2.80GHz processors, with 10 cores per processor.

Overall results. Figure 5 plots the overall performance results. We compare the topology throughput by using two, four, and six computing servers. The y axis is the system throughput (number of tuples per minutes); the x axis denotes the different topologies in comparison: NI stands for the NTM topology, LN-n denotes a LTM topology. In this experiment, the number of bolts for LTM is two, i.e., all LTM topologies have two bolts, B_1 and B_2. A pipeline topology LN-n has the first n NI groups assigned to B_1. Recall that given the three NI groups in Figure 4a, the LN-2 topology has the first two NI groups assigned to B_1 (the rests are in bolt B_2), which results in the right topology in Figure 4b.

The complete LUBM TBox has five NI groups in total. Note that LN-0 places no NI groups in bolt B_1, and has B_2 to check all the NIs.

Throughputs are compared under the same amount of tasks for both topologies. For example, when both methods are set to have six tasks: bolt B_1 of the NTM topology in Figure 3a will have six tasks. Bolts B_1 and B_2 each has three tasks in the LTM topology of Figure 3b. Each task is allocated the same amount of computing resources to guarantee a fair comparison.

Let's consider the results of using two servers. Comparing NI and LN-0, NI has the throughput about twice as much as LN-0 when the task number is six. Since bolt B_1 in LN-0 contains no NIs, its three tasks are only forwarding streams to B_2. The three tasks of B_2 in LN-0 bear the same workload as the six tasks of bolt B_1 in NI. Therefore, the throughput of LN-0 is roughly half of that of NI. This trend can also be observed when there are eight or ten tasks.

We can observe that the throughput of LN-1 greatly improves the one of LN-0. This is because LN-1 distributes the NI-checking workload between B_1 and B_2. Furthermore, the total number of NIs is also reduced, when arranging them in a pipeline. However, the throughputs of NI and LN-1 are roughly the same. It suggests that the way LN-1 distributes the workload cannot outperform NI when having the same amount of tasks. Bolt B_2 in LN-1 is still the bottleneck that limits the throughput.

LN-2 shows the best performance: it outperforms NI by up to 139%. This is because this topology distributes NIs more evenly (there are 2 NI groups assigned to B_1 of LN-2) than LN-0 and LN-1, and shows that LTM can achieve better performance than NTM.

Moving from LN-2 to LN-5 the throughput decreases, since more NI groups are placed onto B_1 than on B_2 and lead to an unbalanced situation. LN-5 has all NI groups assigned to B_1, therefore, its performance is similar to that of LN-0.

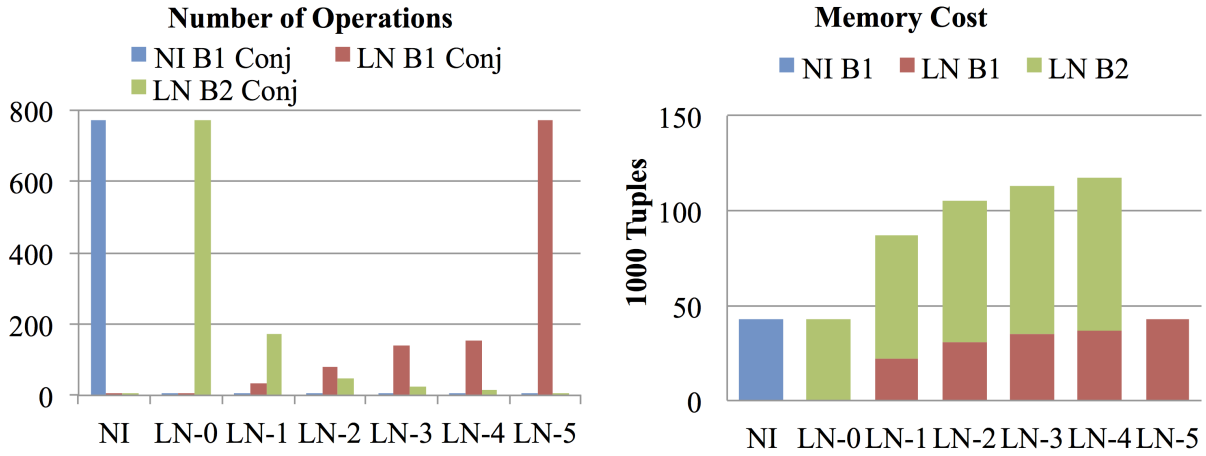
It is worth noting that distributing the same amount of tasks to more servers causes the throughput to decrease. This is due to the higher communication cost across servers.

The change of throughput from LN-0 to LN-5 shows that evenly distributing the NI-checking workload can avoid the bottleneck bolt slowing down the overall performance.

Operations Breakdown and Memory Cost. Figure 6 plots two metrics to further investigate the reasons behind the above results. Figure 6a gives the breakdown of the number of conjunction operations in each bolt. Topologies NI, LN-0, and LN-5 have the same amount of 773 conjunction operations (NIs) executed in one bolts, since the closure of NIs is computed by using all the classes. The comparison among LN-1 to LN-5 shows that the conjunction number of B_1 (green bar) decreases as this number of B_2 increases (red bar). LN-2 has the smallest difference of this number between B_1 and B_2, which suggests the reasons of its best throughput in Figure 5. Regarding the cost of inference operations, we found in our experiments that the ratio between the cost of inference and conjunction (the ratio α in Equation 5) is around 5%. The total number of inference operations is also small (around 20). Therefore, we do not plot this number in the figure.

Figure 6b gives the memory cost (in term of the number of tuples cached). NI, LN-0 and LN-5 have the same memory footprint, since they have similar NI workloads. However, LN-1, LN-2, LN-3 and LN-4 incur higher memory costs than NI, LN-0 and LN-5. As

discussed in Section 4.5, it reflects one shortcoming of LTM that some instances need to be cached multiple times in the pipeline (e.g., In Figure 3b, an instance of *Faculty* is cached at B_1. After B_1 infers it to become *Employee*, the instance needs to cached again in B_2). Furthermore, the memory footprint grows from LN-1 to LN-5. As B_1 handles more classes and inference operations, more instances are going to be cached in B_2. Hence, the total memory cost grows.



(a) Conjunction Operations Breakdown

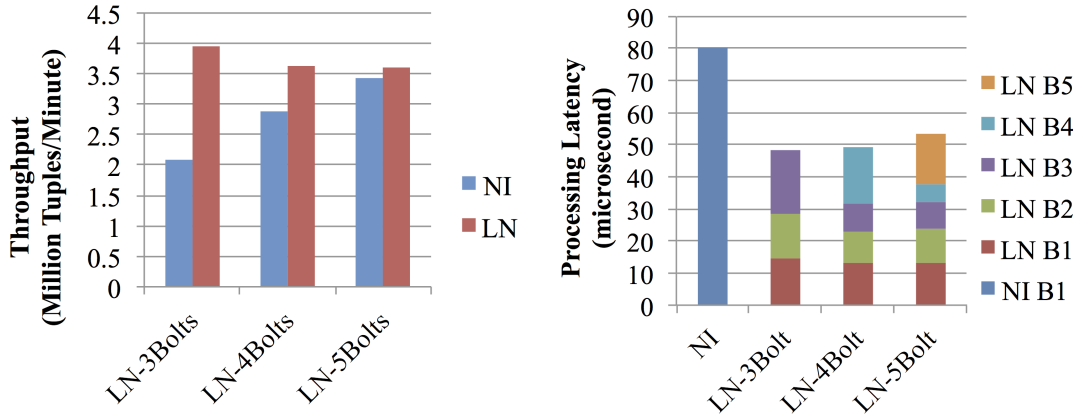
(b) Memory Cost

Fig. 6: Operations breakdown and memory cost.

Results of LTM with multiple bolts. Figure 7a plots the results of comparing a NTM topology with LTM topologies of a different number of bolts (using three servers). We increase the number of bolts for LTM from three to five (each bolt only has one task). Correspondingly, the task number of B_1 in NTM is set to be three, four, and five. We use our cost model to find the best way of assigning NI groups for LTM under different settings. Results in Figure 7a show that the throughput of LTM decreases when using more bolts. This is because the communication cost increases. We can also observe that the throughput of NI topology increases linearly as the number of tasks. This result suggests that when the length of a pipeline topology grows too much, the benefits of reducing NIs might be offset by the communication cost.

Figure 7b plots the breakdown of average processing latency (excluding network latency). It shows that: first, the total processing latency of LTM topologies are all smaller than that of NTM, due to smaller numbers of NIs. Second, when using more bolts of a LTM topology, the processing latency increases. This is caused by the overhead of inference operations and the cost of going through intermediate bolts.

Overall, by combining the results in Figure 5 and 7, we can conclude that LTM improves NTM by evenly distributing the NI-checking workload.



(a) Throughput **(b)** Processing Latency

Fig. 7: LUBM results of a NTM topology and LTM topologies with multiple bolts in the pipeline.

6 Conclusions and Future Work

Our work is inspired by the observation that streaming data are growing in schema complexity. We employ reasoning techniques over streams to detect inconsistencies. While consistency checking is usually studied in the context of static knowledge bases, we first formally adapt this problem for the streaming setting and propose scalable solutions that can be deployed on a DSPE. Based on the DL-lite$_{core}$ logic, our two methods can compile a conceptual model into a processing workflow of a DSPE. The baseline method NTM adapts techniques such as NIs query rewriting to generate continuous queries to assess the consistency. However, NTM involves an excessive number of operations that slow down its performance. To overcome this issue, we propose LTM, where the workload of consistency checking are distributed across a pipeline. This leads to a reduction of the CPU overhead and an improvement of the throughput.

We analyze the trade-offs between NTM and LTM, and propose a cost model to guide the choice between these two methods. Our experiment results show that LTM outperforms NTM by up to 139%, based on the LUBM benchmark.

References

[1] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, February 2012.

[2] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *PVLDB*, 6(11):1033–1044, 2013.

[3] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. Ep-sparql: A unified language for event processing and stream reasoning. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 635–644, New York, NY, USA, 2011. ACM.

[4] Carlos Buil Aranda, Marcelo Arenas, Óscar Corcho, and Axel Polleres. Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. *J. Web Sem.*, 18(1):1–17, 2013.

[5] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '99, pages 68–79, New York, NY, USA, 1999. ACM.

[6] Alessandro Artale, Diego Calvanese, Roman Kontchakov, and Michael Zakharyaschev. The dl-lite family and relations. *CoRR*, abs/1401.3487, 2014.

[7] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Comput. Netw.*, 54(15):2787–2805, October 2010.

[8] Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the el envelope. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, IJCAI'05, pages 364–369, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc.

[9] Franz Baader, Carsten Lutz, and Sebastian Brandt. Pushing the EL Envelope Further. In *OWLED (Spring)*, volume 496 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[10] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, pages 350–, Washington, DC, USA, 2004. IEEE Computer Society.

[11] Kenneth Baclawski, Mieczyslaw M. Kokar, Richard J. Waldinger, and Paul A. Kogut. Consistency checking of semantic web ontologies. pages 454–459, 2002.

[12] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-SPARQL: A Continuous Query Language for RDF Data Streams. *International Journal of Semantic Computing*, (1):3–25, 2010.

[13] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Incremental reasoning on streams and rich background knowledge. In *Proceedings of the 7th International Conference on The Semantic Web:*

Research and Applications - Volume Part I, ESWC'10, pages 1–15, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying RDF streams with C-SPARQL. *SIGMOD Record*, 39(1):20–26, 2010.

[15] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

[16] Elena Botoeva, Alessandro Artale, and Diego Calvanese. Query Rewriting in DL-Lite$_{horn}^{(HN)}$. In *Description Logics*, volume 573 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2010.

[17] I.T. Bowman and G.N. Paulley. Join enumeration in a memory-constrained environment. In *Proc. ICDE*, pages 645–654, 2000.

[18] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair J. G. Gray. Enabling ontology-based access to streaming data sources. In *Proceedings of the 9th International Semantic Web Conference on The Semantic Web - Volume Part I*, ISWC'10, pages 96–111, Berlin, Heidelberg, 2010. Springer-Verlag.

[19] Jean-Paul Calbimonte, Hoyoung Jeung, Óscar Corcho, and Karl Aberer. Enabling query technologies for the semantic sensor web. *Int. J. Semantic Web Inf. Syst.*, 8(1):43–63, 2012.

[20] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[21] Jiaoyan Chen, Huajun Chen, Zhaohui Wu, Daning Hu, and Jeff Z. Pan. Forecasting smog-related health hazard based on social media and physical sensor. *Inf. Syst.*, 64(C):281–291, March 2017.

[22] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[23] R. Cyganiak, D. Wood, and M. Lanthaler. Rdf 1.1 concepts and abstract syntax, 2014.

[24] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 40–51, New York, NY, USA, 2003. ACM.

[25] Patrick De Boer and Abraham Bernstein. Efficiently identifying a well-performing crowd process for a given problem. In *20th ACM Conference on Computer-Supported Cooperative Work and Social Computing (CSCW 2017)*, Portland, OR, FEB 2017.

[26] Soheila Dehghanzadeh, Daniele Dell'Aglio, Shen Gao, Emanuele Della Valle, Alessandra Mileo, and Abraham Bernstein. Approximate continuous query answering over streams and dynamic linked data sets. In *Proceedings of the 15th International Conference on Engineering the Web in the Big Data Era - Volume 9114*, ICWE 2015, pages 307–325, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[27] Soheila Dehghanzadeh, Josiane Parreira, Marcel Karnstedt, Jürgen Umbrich, Manfred Hauswirth, and Stefan Decker. Optimizing SPARQL query processing on dy-

namic and static data based on query time/freshness requirements using materialization. In *JIST*, 2014.

[28] Daniele Dell' Aglio, Emanuele Della Valle, Frank van Harmelen, and Abraham Bernstein. Stream reasoning: A survey and outlook. *Data Science*, 1(1-2):59–83, 2017.

[29] Daniele Dell' Aglio, Emanuele Della Valle, Frank van Harmelen, and Abraham Bernstein. Stream reasoning: A survey and outlook. *Data Science*, pages 1–24, 2017.

[30] Daniele Dell'Aglio, Emanuele Della Valle, Jean-Paul Calbimonte, and Oscar Corcho. Rsp-ql semantics: A unifying query model to explain heterogeneity of rdf stream processing systems. *Int. J. Semant. Web Inf. Syst.*, 10(4):17–44, October 2014.

[31] Yanlei Diao, Neil Immerman, and Daniel Gyllstrom. Sase+: An agile language for kleene closure over event streams. Technical report, University of Massachusetts Amherst, Department of Computer Science, 2008.

[32] Lorenz Fischer, Shen Gao, and Abraham Bernstein. Machines tuning machines: Configuring distributed stream processors with bayesian optimization. In *Proceedings of the 2015 IEEE International Conference on Cluster Computing*, CLUSTER '15, pages 22–31, Washington, DC, USA, 2015. IEEE Computer Society.

[33] Stéphane Gançarski, Hubert Naacke, Esther Pacitti, and Patrick Valduriez. The leganet system: Freshness-aware transaction routing in a database cluster. *Info. Systems*, 2005.

[34] Shen Gao, Daniele Dell'Aglio, Soheila Dehghanzadeh, Abraham Bernstein, Emanuele Della Valle, and Alessandra Mileo. Planning ahead: Stream-driven linked-data access under update-budget constraints. In *The 15th International Semantic Web Conference*, Heidelberg, 2016.

[35] Shen Gao, Thomas Scharrenbach, and Abraham Bernstein. The clock data-aware eviction approach: Towards processing linked data streams with limited resources. In *The 11th Extended Semantic Web Conference*, Lecture Notes in Computer Science. Springer, MAY 2014.

[36] Shen Gao, Thomas Scharrenbach, Jörg-Uwe Kietz, and Abraham Bernstein. Running out of bindings? integrating facts and events in linked data stream processing. In *4th International Workshop on Ordering and Reasoning*, Aachen, Germany, OCT 2015. s.n.

[37] Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. HermiT: An OWL 2 Reasoner. *J. Autom. Reasoning*, 53(3):245–269, 2014.

[38] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 1487–1495, New York, NY, USA, 2017. ACM.

[39] Hongfei Guo, Per Larson, and Raghu Ramakrishnan. Caching with "good enough" currency, consistency, and completeness. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 457–468. VLDB Endowment, 2005.

[40] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semant.*, 3(2-3):158–182, October 2005.

[41] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language. Technical report, The World Wide Web Consortium (W3C), 2011.

[42] Souleiman Hasan, Sean O'Riain, and Edward Curry. Towards unified and native enrichment in event processing systems. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 171–182, New York, NY, USA, 2013. ACM.

[43] Annika Hinze, Kai Sachs, and Alejandro Buchmann. Event-based applications and enabling technologies. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 1:1–1:15, New York, NY, USA, 2009. ACM.

[44] Yuanzhen Ji, Zbigniew Jerzak, Anisoara Nica, Gergor Hackenbroich, and Christof Fetzer. Optimization of continuous queries in federated database and stream processing systems. pages 403–422, 2015.

[45] Tobias Käfer, Jürgen Umbrich, Aidan Hogan, and Axel Polleres. Towards a dynamic linked data observatory. *LDOW at WWW*, 2012.

[46] Jörg-Uwe Kietz, Thomas Scharrenbach, Lorenz Fischer, Abraham Bernstein, and Khoa Nguyen. TEF-SPARQL: The DDIS query-language for time annotated event and fact Triple-Streams. Technical report, University of Zurich, Department of Informatics, 2013.

[47] R. Kontchakov, C. Lutz, D. Toman, F. Wolter, and M. Zakharyaschev. The combined approach to query answering in dl-lite. In *Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning*, KR'10, pages 247–257. AAAI Press, 2010.

[48] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[49] Alexandros Labrinidis and Nick Roussopoulos. Exploring the tradeoff between performance and data freshness in database-driven web servers. volume 13, pages 240–255, Secaucus, NJ, USA, September 2004. Springer-Verlag New York, Inc.

[50] Günter Ladwig and Thanh Tran. Sihjoin: Querying remote and local linked data. In *Proceedings of the 8th Extended Semantic Web Conference on The Semantic Web: Research and Applications - Volume Part I*, ESWC'11, pages 139–153, Berlin, Heidelberg, 2011. Springer-Verlag.

[51] Danh Le Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *ISWC 2011, Proceedings, Part I*, pages 370–388, 2011.

[52] Freddy Lécué and Jeff Z. Pan. Consistent knowledge discovery from evolving ontologies. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 189–195. AAAI Press, 2015.

[53] Rubao Lee and Zhiwei Xu. Exploiting stream request locality to improve query throughput of a data integration system. *IEEE Trans. on Computers*, 58(10):1356–1368, 2009.

[54] Domenico Lembo and Marco Ruzzi. Consistent query answering over description logic ontologies. In *Proceedings of the 1st International Conference on Web Reasoning and Rule Systems*, RR'07, pages 194–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[55] John D. C. Little. A proof for the queuing formula: L= λ w. *Operations Research*, 9(3):pp. 383–387, 1961.

[56] Alessandro Margara, Jacopo Urbani, Frank van Harmelen, and Henri Bal. Streaming the web: Reasoning over dynamic data. *J. Web Sem*, 25:24–44, 2014.

[57] Amélie Marian and Jérôme Siméon. Projecting xml documents. In *Proc. VLDB*, pages 213–224, 2003.

[58] Gavin Mendel-Gleason, Kevin Feeney, and Rob Brennan. Ontology consistency and instance checking for real world linked data. 1376, 2015.

[59] Alejandro Metke-Jimenez and Michael Lawley. Snorocket 2.0: Concrete domains and concurrent classification. In Samantha Bail, Birte Glimm, Rafael S. Gonçalves, Ernesto Jiménez-Ruiz, Yevgeny Kazakov, Nicolas Matentzoglu, and Bijan Parsia, editors, *ORE*, volume 1015 of *CEUR Workshop Proceedings*, pages 32–38. CEUR-WS.org, 2013.

[60] Gabriela Montoya, Maria-Esther Vidal, Oscar Corcho, Edna Ruckhaus, and Carlos Buil-Aranda. Benchmarking federated sparql query engines: Are existing testbeds enough? In *ISWC*, pages 313–324, 2012.

[61] Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz. OWL 2 Web Ontology Language Profiles (Second Edition). W3c Recommendation, W3C, 2012.

[62] K. V. M. Naidu, Rajeev Rastogi, Scott Satkin, and Anand Srinivasan. Memory-constrained aggregate computation over data streams. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 852–863, Washington, DC, USA, 2011. IEEE Computer Society.

[63] Khoa Nguyen, Thomas Scharrenbach, and Abraham Bernstein. Eviction strategies for semantic flow processing. In *Proceedings of the 9th International Conference on Scalable Semantic Web Knowledge Base Systems - Volume 1046*, SSWS'13, pages 66–80, Aachen, Germany, Germany, 2013. CEUR-WS.org.

[64] Heiko Paulheim and Aldo Gangemi. Serving dbpedia with dolce — more than just adding a cherry on top. pages 180–196, 2015.

[65] Heiko Paulheim and Heiner Stuckenschmidt. Fast approximate a-box consistency checking using machine learning. In *Proceedings of the 13th International Conference on The Semantic Web. Latest Advances and New Domains - Volume 9678*, pages 135–150, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[66] Héctor Pérez-Urbina, Boris Motik, and Ian Horrocks. A comparison of query rewriting techniques for dl-lite. In *Proceedings of the 22nd International Workshop on Description Logics (DL 2009), Oxford, UK, July 27-30, 2009*, 2009.

[67] D. Puiu, P. Barnaghi, R. Tönjes, D. Kümper, M. I. Ali, A. Mileo, J. Xavier Parreira, M. Fischer, S. Kolozali, N. Farajidavar, F. Gao, T. Iggena, T. L. Pham, C. S. Nechifor, D. Puschmann, and J. Fernandes. Citypulse: Large scale data analytics framework for smart cities. *IEEE Access*, 4:1086–1108, 2016.

[68] Yuan Ren and Jeff Z. Pan. Optimising ontology stream reasoning with truth maintenance system. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 831–836, New York, NY, USA, 2011. ACM.

[69] Yuan Ren, Jeff Z. Pan, Isa Guclu, and Martin J. Kollingbaum. A combined approach to incremental reasoning for EL ontologies. In *Web Reasoning and Rule Systems - 10th International Conference, RR 2016, Aberdeen, UK, September 9-11, 2016, Proceedings*, pages 167–183, 2016.

[70] Mikko Rinne, Monika Solanki, and Esko Nuutila. Rfid-based logistics monitoring with semantics-driven event processing. In *DEBS*, pages 238–245, 2016.

[71] Stefan Schulz, Ronald Cornet, and Kent Spackman. Consolidating snomed ct's ontological commitment. *Appl. Ontol.*, 6(1):1–11, January 2011.

[72] Mohamed Sharaf, Panos Chrysanthis, and Alexandros Labrinidis. Preemptive rate-based operator scheduling in a data stream management system. In *AICCSA*, pages 46–59, 2005.

[73] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS'12, pages 2951–2959. Curran Associates Inc., USA, 2012.

[74] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 595–604, New York, NY, USA, 2008. ACM.

[75] Heiner Stuckenschmidt, Stefano Ceri, Emanuele Della Valle, and Frank van Harmelen. Towards expressive stream reasoning. In Karl Aberer, Avigdor Gal, Manfred Hauswirth, Kai-Uwe Sattler, and Amit P. Sheth, editors, *Semantic Challenges in Sensor Networks*, number 10042 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2010. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[76] Martin Suda, Christoph Weidenbach, and Patrick Wischnewski. On the saturation of yago. In *Proceedings of the 5th International Conference on Automated Reasoning*, IJCAR'10, pages 441–456, Berlin, Heidelberg, 2010. Springer-Verlag.

[77] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 309–320. VLDB Endowment, 2003.

[78] Kia Teymourian and Adrian Paschke. Plan-based semantic enrichment of event streams. In *11th Extended Semantic Web Conference (ESWC 2014)*, Crete, Greece, 2014.

[79] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.

[80] Jacopo Urbani. *On Web-scale Reasoning.* PhD thesis, Vrije Universiteit Amsterdam, 2013.

[81] J.C. Whittier, Silvia Nittel, and Iranga Subasinghe. Real-time earthquake monitoring with spatio-temporal fields. 08 2017.

[82] Jiewen Wu and Freddy Lecue. Towards consistency checking over evolving ontologies. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '14, pages 909–918, New York, NY, USA, 2014. ACM.

[83] Shima Zahmatkesh, Emanuele Della Valle, and Daniele Dell'Aglio. When a FILTER makes the difference in continuously answering SPARQL queries on streaming and quasi-static linked data. In *Web Engineering - 16th International Conference, ICWE 2016, Lugano, Switzerland, June 6-9, 2016. Proceedings*, pages 299–316, 2016.

[84] Shima Zahmatkesh, Emanuele Della Valle, and Daniele Dell'Aglio. Using rank aggregation in continuously answering SPARQL queries on streaming and quasi-static linked data. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017*, pages 170–179, 2017.

[85] Ying Zhang, Minh-Duc Pham, Óscar Corcho, and Jean-Paul Calbimonte. Srbench: A streaming RDF/SPARQL benchmark. In *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I*, pages 641–657, 2012.

[86] Zhenyun Zhuang, Tao Feng, Yi Pan, Haricharan Ramachandra, and Badri Sridharan. Effective multi-stream joining in apache samza framework. pages 267–274, June 2016.

Curriculum Vitæ

SHEN GAO

Room BIN 2.D.29, Department of Informatics, University of Zurich
Email: shengao@ifi.uzh.ch Tel: (+41) 787298219

Working & Teaching Experience

Google Jun 2016 - Sep 2016
Software Engineer Internship Zurich, Switzerland

Teaching Assistant, University of Zurich Sep 2013 - Now
Advisor for two master theses/projects and two bachelor theses. Switzerland
Assistant for the Distributed Systems course, which involved teaching tutorials and designing assignments about Hadoop, Pig, and Spark.

Education

Ph.D. Student in Computer Science Sep 2013 - Present
University of Zurich Switzerland
Supervisor: Prof. Abraham Bernstein

Master of Philosophy in Computer Science Sep 2010 - Mar 2013
Hong Kong Baptist University Hong Kong
Supervisor: Prof. Jianliang Xu GPA:4.0/4.0

Bachelor of Science in Computing Studies (Information Systems) Sep 2006 - Jul 2010
Hong Kong Baptist University Hong Kong
First-Class Honours GPA:3.64/4.0

Visiting Student
Nanyang Technological University, Singapore Mar 2013 - Jun 2013
University of Kaiserslautern, Germany Aug 2012
University of Leeds, United Kingdom Sep 2008 - Feb 2009

Research Projects

RDF Stream Processing (RSP)
Real-time processing of massive, dynamically generated stream-data has become increasingly popular in the world of Linked Open Data. My current research problems include: scalable and efficient joining of stream and remote static data [2, 4, 5, 6], handling unstable stream income rate [7], and designing new syntax semantics for querying streams [3]. As a contributor, I integrated some of my work into a stream reasoning engine, C-SPARQL `https://bitbucket.org/soheilade/newcsparql/commits/all`, and a SPARQL engine, Jena ARQ `https://bitbucket.org/dellaglio/jena/commits/all`.

Exploiting Phase Change Memory (PCM) for Future Computer Systems
PCM is a promising next-generation memory technology. Its superb features, such as non-volatility and bit-alterability are calling for a revisit on traditional database system design assumptions. To exploit those features, I designed a new logging system–PCMLogging, which is an alternative to the conventional WAL log [10] and a new checkpointing system for High Performance Computing (HPC) systems [9]. Both systems have been implemented in a state-of-the-art disk simulator, Disksim.

Data Management on Flash Disks
Flash disks have been widely deployed on mobile devices. In my bachelor study, I was involved in the design of new buffer replacement algorithms for databases on flash disks and implemented them in real systems, such as PostgreSQL. [11, 14]

Programming & System Skills

Working knowledge: Java, C
Basic knowledge: C++, C#, Python, Objective C
Experience with: Amazon Cloud Services, Microsoft Azure, Google App Engine

Language Skills

Chinese	Native	***Cantonese***	Professional working proficiency
English	Professional working proficiency	***German***	Basic

Publications

[1] **Shen Gao**, Daniele Dell'Aglio, Jeff Z. Pan and Abraham Bernstein. "Distributed Stream Consistency Checking" *Submitted to IEEE International Conference on Big Data* ***(BigData '17)***

[2] **Shen Gao**, Daniele Dell'Aglio, Soheila Dehghanzadeh, Abraham Bernstein, Emanuele Della Valle and Alessandra Mileo. "Planning Ahead: Stream-Driven Linked-Data Access under Update-Budget Constraints" *International Semantic Web Conference* ***(ISWC '16)*** **(Student Travel Award)**

[3] **Shen Gao**, Thomas Scharrenbach, Jörg-Uwe Kietz, Abraham Bernstein. "Running out of Bindings? Integrating Facts and Events in Linked Data Stream Processing" *International Workshop on Ordering and Reasoning* ***(OrdRing '15)***.

[4] Lorenz Fischer, **Shen Gao**, Abraham Bernstein. "Machines Tuning Machines: Configuring Distributed Stream Processors with Bayesian Optimization" *2015 IEEE International Conference on Cluster Computing* ***(CLUSTER '15)***. **(Best Paper Candidate)**

[5] Soheila Dehghanzadeh, Daniele Dell'Aglio, **Shen Gao**, Emanuele Della Valle, Alessandra Mileo and Abraham Bernstein. "Approximate Continuous Query Answering over Streams and Dynamic Linked Data Sets." *International Conference on Web Engineering* ***(ICWE '15)***.

[6] Soheila Dehghanzadeh, Daniele Dell'Aglio, **Shen Gao**, Emanuele Della Valle, Alessandra Mileo and Abraham Bernstein. "Online View Maintenance for Continuous Query Evaluation." *International World Wide Web Conference* ***(WWW '15)***. (Poster)

[7] **Shen Gao**, Thomas Scharrenbach and Abraham Bernstein. "The CLOCK Data-Aware Eviction Approach: Towards Processing Linked Data Streams with Limited Resources." *Extended Semantic Web Conference* ***(ESWC '14)***.

[8] Michael Feldman, **Shen Gao**, Marc Novel, Katerina Papaioannou and Abraham Bernstein. " SHAX: The Semantic Historical Archive eXplorer." *International Semantic Web Conference* ***(ISWC '14)***. (Poster)

[9] **Shen Gao**, Bingsheng He, and Jianliang Xu. "Real-Time In-Memory Checkpointing for Future Hybrid Memory Systems." *International Conference on Supercomputing* ***(ICS '15)***.

[10] **Shen Gao**, Jianliang Xu, Theo Härder, Bingsheng He, Byron Choi, Haibo Hu. "PCMLogging: Optimizing Transaction Logging and Recovery Performance with PCM." *IEEE Transactions on Knowledge and Data Engineering* ***(IEEE Trans. TKDE)***

[11] Saitung On, **Shen Gao**, Bingsheng He, Ming Wu, Qiong Luo, and Jianliang Xu. "FD-Buffer: A Buffer Manager for Databases on Flash Memory." *IEEE Transactions on Computers****(IEEE Trans. TC)***

[12] **Shen Gao**, Jianliang Xu, Bingsheng He, Byron Choi, and Haibo Hu. "PCMLogging: Reducing Transaction Logging Overhead with PCM." *ACM Conference on Information and Knowledge Management* ***(ACM CIKM '11)***.(Poster)

[13] Ziwei Yang, **Shen Gao**, Jianliang Xu, and Byron Choi. "Authentication of Range Query Results in MapReduce Environments." *ACM Workshop on Cloud Data Management* ***(CloudDB '11)***.

[14] **Shen Gao**, Yu Li, Jianliang Xu, Byron Choi, and Haibo Hu. "DigestJoin: Expediting Joins on Solid-State Drives." *International Conference on Database Systems for Advanced Applications* ***(DASFAA '10)***.

Others

Attended the Advanced Statistics and Data Mining Summer School Jun 2015

First Runner-up, IBM DB2 UDB Inter-University Programming Contest Feb 2008
Participants included 28 teams from universities in Hong Kong and Macau

China Soong Ching Ling Foundation Zhi Yuan Scholarship Aug 2006
30-40 honored out of 2000+ mainland China undergraduate students in Hong Kong